Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Realidad Aumentada: Compartir Objetos Virtuales

TITULACIÓN: Grado en Ingeniería Sistemas de Telecomunicación

AUTOR: Guillermo Arribasplata Huaman

DIRECTOR: Dolors Royo Vallés

FECHA:

Resumen

Desde la EETAC, se intenta desarrollar una aplicación con la que se pueda llevar a cabo un control aéreo, mediante un despliegue de drones y el uso de una aplicación basada en la tecnología de realidad aumentada.

El objetivo de este proyecto, es crear una aplicación que brinde un entorno para controladores aéreos donde puedan interactuar entre ellos, crear objetos virtuales en base a la información recogida por los drones, que los objetos virtuales sean capaces de responder a una serie de funciones y que dichas respuestas sean vistas por otros controladores aéreos.

En este proyecto se propone la manera de crear un entorno de visualización compartida para los controladores aéreos, dicho entorno debe permitir la creación, gestión, manipulación de los objetos virtuales por parte de los controladores aéreos.

A partir de aquí, lo siguiente fue hacer uso de la documentación proporcionada por el sitio web HoloLens Academy para comprender el funcionamiento del dispositivo.

En el proceso de documentación se observó que Microsoft Windows ofrece un paquete de scripts que permite acelerar el desarrollo de aplicaciones para el dispositivo HoloLens haciendo uso del motor de Unity3D, motor que se usa para la realización de videojuegos para diferentes dispositivos.

Así pues una vez que se adquirió el conocimiento, lo siguiente fue analizar el código del paquete de scripts provisto por Microsoft Windows para proceder a construir una aplicación que cumpliera con los objetivos funcionales de nuestro proyecto.

Este proyecto hizo uso de los gestos básicos reconocidos por HoloLens, del lenguaje de programación C # y del entorno de edición visual studio 2017.

Las pruebas realizadas se presentarán en los resultados del capítulo de este mismo documento.

Title: Augmented Reality: Share Virtual Objects

Author: Guillermo Arribasplata Huaman

Director: Dolors Royo Vallés

Date:

Overview

From the EETAC, trying to develop an application that can carry out an air control, through deployment of drones and the use of an application based on augmented reality technology.

The objective of this project is to create an application that provides an environment for air traffic controllers where they can interact with each other, create virtual objects based on the information collected by the drones, that the virtual objects are able to respond to a series of functions and that these answers are seen by other air traffic controllers.

In this project we propose the way to create a shared viewing environment for air traffic controllers, this environment should allow the creation, management, manipulation of virtual objects by air traffic controllers.

From here, the following was to make use of the documentation provided by the HoloLens Academy website to understand the operation of the device.

In the documentation process it was observed that Microsoft Windows offers a script package that allows to accelerate the development of applications for the HoloLens device using the Unity3D engine, which is used to make video games for different devices.

So once the knowledge was acquired, the next thing was to analyze the code of the script package provided by Microsoft Windows to proceed to build an application that fulfilled the functional objectives of our project.

This project made use of the basic gestures recognized by HoloLens, the C # programming language and the visual editing studio 2017 environment.

The tests carried out will be presented in the results of the chapter of this same document.

Agradecimientos

Quisiera aprovechar estas líneas para mostrar mi más sincero agradecimiento a la profesora y directora de este proyecto Dolors Royo Vallés que siempre mostro una gran disposición por ayudarme con el desarrollo de este proyecto, desde el momento en que le presente mi propuesta.

También quiero hacer extensiva este agradecimiento a mis compañeros Eric Soria Martinez, Azeez Olusegun Odumosu y Alberto Kevin Rodas con los cuales he ido compartiendo experiencias a lo largo de estos años y que siempre han sido de ayuda.

Por ultimo agradecer a mi familia y seres queridos que siempre estuvieron apoyándome.

A todos ellos, Muchas gracias.

ÍNDICE

CAPITULO 1. INTRODUCCIÓN

1.1 Justificación del proyecto

En 2016 aparece la aplicación Pokemon-Go, videojuego que hace renacer nuevamente el interés de algunas empresas por la realidad aumentada, tecnología que ya existía años atrás pero que el ciudadano de a pie desconocía.

La aparición de nuevas aplicaciones y dispositivos que hagan uso de la realidad aumentada se vuelve más frecuente, sobre todo en el sector de los videojuegos, y a partir de aquí se podría decir que la realidad aumentada se volvió una tendencia tecnológica nuevamente.

Así que el proyecto surge del interés por aprender sobre el uso de software destinado a videojuegos y de que la EETAC pretende crear una aplicación que permita el control aéreo en base a un despliegue de drones y la realidad aumentada.

1.2 Objetivo del proyecto

El proyecto tiene como objetivo crear una aplicación que haga uso del dispositivo de realidad aumentada HoloLens de Microsoft Windows, donde la aplicación sea capaz de brindar un entorno de visualización compartida sobre objetos virtuales, así como la posibilidad de observar su información y el interactuar sobre ellos.

Permitiendo así crear aplicaciones más complejas a partir de las herramientas descubiertas y de esta idea básica.

Este proyecto no tiene como objetivo, servir de soporte para la mejora de la presentación y visualización de los objetos virtuales en el mundo real.

CAPITULO 2. ESPECIFICACIONES Y REQUERIMIENTOS

En este capítulo hablaremos de los requisitos a nivel funcional que nuestra aplicación debe de cumplir como objetivo.

Así pues haremos referencia a las herramientas seleccionadas para el desarrollo de nuestra aplicación. Esta selección está basada en dos aspectos importante.

- Los requerimientos necesarios para utilizar un emulador HoloLens, debido a la dificultad de acceder a un dispositivo propio.
- La importación del paquete de herramientas HoloToolKit, que nos ayudara a simular el comportamiento de unas HoloLens dentro del entorno de Unity.

2.1 Requisitos

Para poder desarrollar este proyecto, se ha considerado necesario definir una serie de propiedades y expectativas sobre el uso del entorno.

- Se creara un entorno compartido para los diferentes controladores aéreos.
- Los controladores aéreos utilizaran el entorno con la finalidad de crear objetos virtuales.
- Los controladores aéreos compartirán la visualización de algunos objetos virtuales.
- Los controladores aéreos serán capaces de manipular los objetos virtuales y que dicha manipulación sea vista por otros controladores aéreos.

2.1.1 Requisitos no funcionales

En este apartado hablaremos en términos generales, de las herramientas que se ha utilizado para este proyecto (ilustración 1).

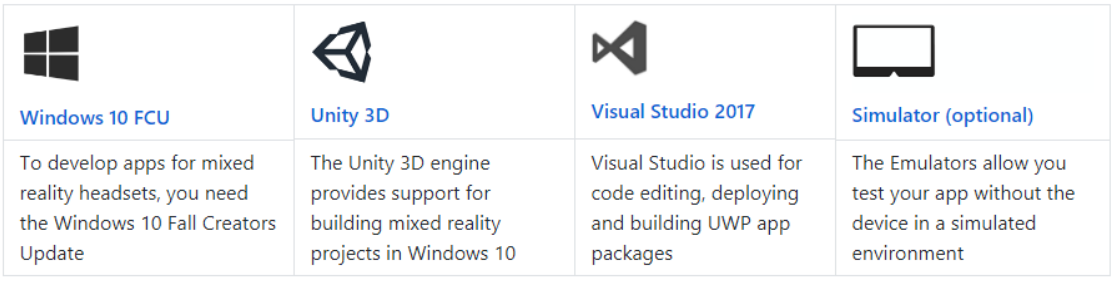

Ilustración 1. Software requerido. [2]

2.1.2.1 Sistema operativo

Este proyecto está pensado para funcionar en la familia de sistemas operativos Microsoft Windows 10 SDK, este SDK también es compatible con Windows 8.1, Windows 8, Windows7.

Este proyecto se ha realizado en Windows 10 Pro 64 bits, sistema que soporta Hyper-V. Pensando en la posibilidad de no tener a disposición el dispositivo HoloLens y que para ello se necesitara hacer uso de un emulador de HoloLens para ver los progresos del proyecto.

2.1.2.2 Unity y C#

Este proyecto se realizó pensando que el dispositivo a utilizar por parte de los controladores aéreos, son las HoloLens y Unity nos permite fácilmente portar el paquete de scripts *Mixed Reality Toolkit* (herramientas de realidad mixta), que nos permite acelerar el desarrollo de aplicaciones dirigida a las gafas HoloLens.

Este paquete nos ofrece ciertos niveles de abstracción (ilustración 2) a la hora de desarrollar una aplicación.

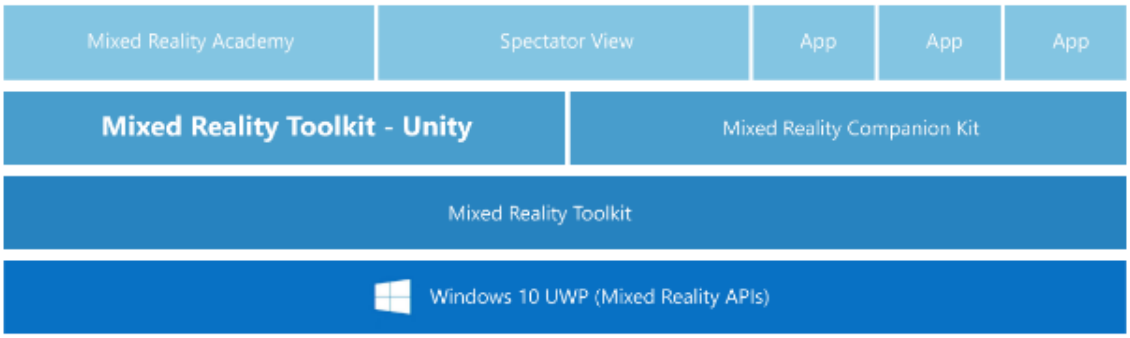

Ilustración 2. Mixed Reality Toolkit – Unity, niveles de abstracción. [2]

Junto a Unity hemos decidido utilizar C#, que es el lenguaje del cual contaba ya con un conocimiento básico, además de que es el lenguaje mayoritario para el desarrollo de aplicaciones en Unity (ilustración 3) y más importante aún es que el paquete de scripts *Mixed Reality Toolkit* está realizado en C#.

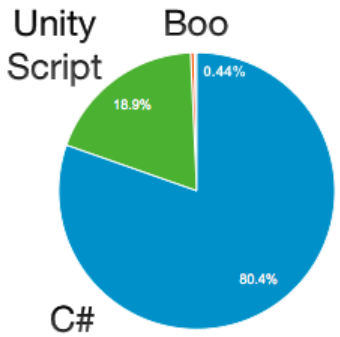

Ilustración 3. Lenguajes utilizados en Unity3D. [3]

2.1.2.3 Visual Studio 2017

Se utiliza para la edición de código, implementación y creación de paquetes de aplicaciones, En este proyecto en concreto se usó la versión Visual Studio 2017 Profesional dado que permite la compilación de las aplicaciones en el emulador

HoloLens, aun así, según la documentación que ofrece Microsoft, todas las versiones de Visual Studio 2017 son compatibles (incluida *Comunity*), si bien la actualización 3 de Visual Studio 2015 aun es compatible, recomienda las versiones 2017.

2.1.2.3 Hololens Emulator – Mixed Reality Simulator

El emulador es una máquina virtual que permite ejecutar aplicaciones holográficas de Windows.

Este Incluye una imagen HoloLens virtual (ilustración 4) que ejecuta la última versión del sistema operativo holográfico de Windows, en este proyecto en concreto se ha utilizado el compilador 10.0.14393.1358.

Para que esto funcione correctamente, el sistema operativo debe soportar Hyper-V (Windows 10 Pro 64bits, *Enterprise* o *Education*), el *Home* no es compatible.

Ilustración 4. HoloLens Emulator. [1]

2.1.2 Requisitos funcionales

En este apartado definiremos las acciones que la aplicación ha de permitir realizar a los controladores aéreos y la plasmaremos como caso de usos.

- Caso de uso para inicio de sesión.
- Caso de uso de acción dentro de una sesión.

Los casos de uso están definidos de esta manera debido a la tecnología que nos ofrece Unity.

2.1.1.1 Inicio de sesión

Los controladores aéreos pueden decidir qué tipo de rol asumir, si ser quien genera la sesión o quien se una a alguna sesión existente (ilustración 5 y 6).

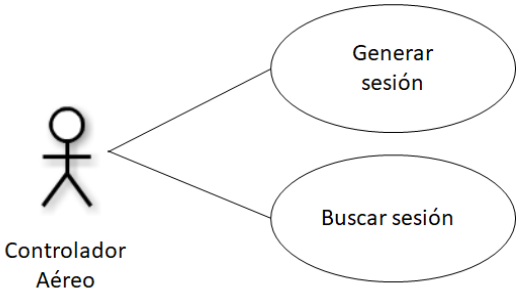

Ilustración 5. Caso de uso para inicio de sesión.

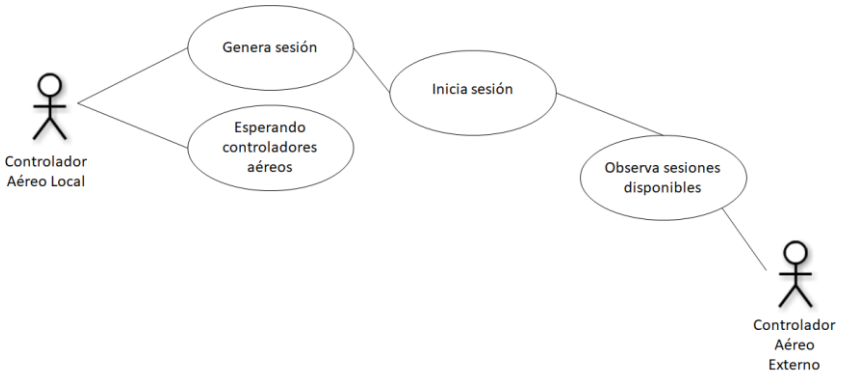

Ilustración 6. Caso de uso para inicio de sesión

En el caso de que un controlador aéreo asuma generar la sesión, este será el controlador aéreo local (servidor) esperando que los demás controladores aéreos externos se conecten a é (clientes).

2.1.1.1 Acción dentro de la sesión.

Los controladores aéreos dentro de una sesión deben ser capaces de realizar las siguientes acciones sobre los diferentes objetos virtuales a los cuales denominamos avión (ilustración 7).

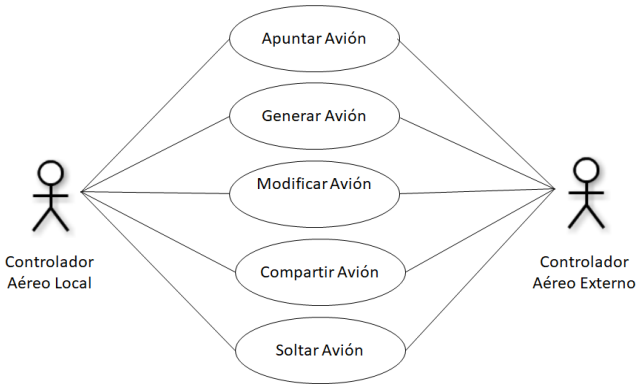

Ilustración 7. Caso de uso dentro de la sesión

2.2 Arquitectura lógica

El sistema seguirá una estructura cliente-servidor (ilustración 8), dado que es la que nos ofrece la API Unet de Unity, esto nos da la posibilidad de separar la

lógica correspondiente a cada rol, y así asegurar su compresión para futuras aplicaciones.

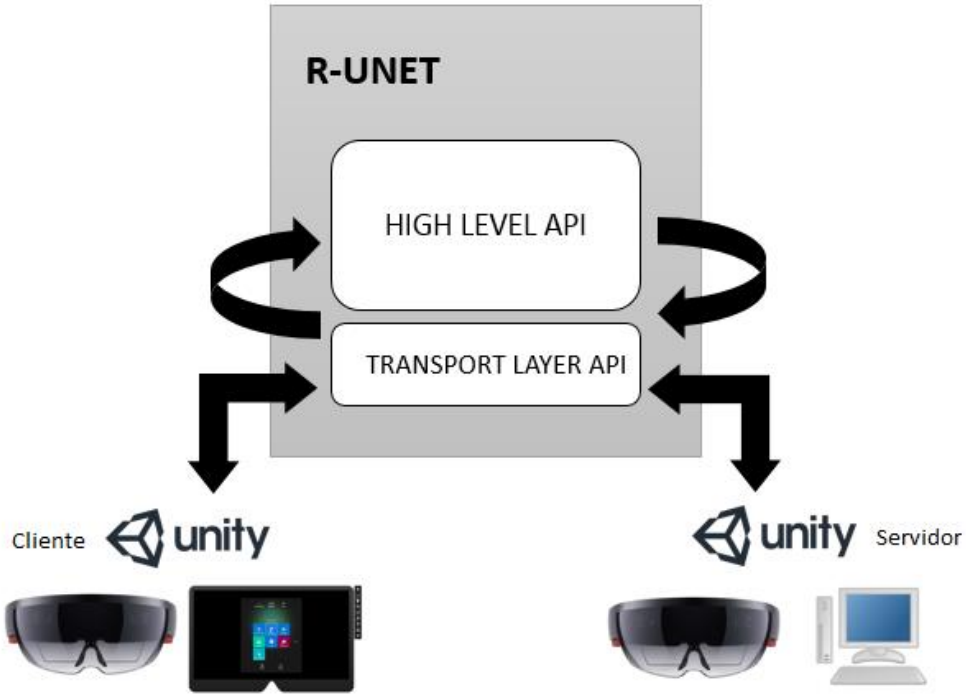

Ilustración 8. Arquitectura lógica del proyecto

CAPITULO 3. HOLOLENS Y LA REALIDAD AUMENTADA

En este capítulo hablaremos sobre conceptos básicos a tener en cuenta sobre la realidad aumentada, así como el impacto de esta tecnología en muchos sectores de la industria.

También hacemos mención de los controles básicos, los cuales hacemos uso dentro de nuestra aplicación.

3.1 ¿Qué es realidad aumentada?

El término realidad aumentada (RA) se usa para definir la tecnología que nos permite superponer capas de información al mundo real en tiempo real; Ya sea con imágenes, marcadores e información generada virtualmente, lo cual enriquece nuestra percepción de la realidad.

Se crea de esta manera un entorno donde coexisten los objetos virtuales con la realidad misma, ofreciendo así una experiencia tal que incluso un usuario puede llegar a percibir los objetos virtuales parte de su realidad cotidiana.

Durante sus primeros días, la RA prometía cambiar la forma en que se interactuase con el mundo real, se veía como una fuerza para acelerar el progreso, permitir aquello llamado como el “sexto sentido digital” y divertirse más allá de la pantalla; La RA era lo mismo que decir humanidad aumentada. Pero faltaban las plataformas que permitiesen llegar a las personas y sus problemas cotidianos, para así conseguir concienciar al público general sobre su potencial; Hasta que aparecieron los teléfonos inteligentes.

Desde un punto de vista pragmático, la realidad aumentada es un servicio compuesto por 4 elementos básicos.

- **Elemento que capture las imágenes de la realidad que están viendo los usuarios**, basta con ello una cámara de las que están presente en los teléfonos inteligentes y ordenadores.
- **Elemento sobre el que proyectar la mezcla de las imágenes reales con las imágenes sintetizadas**, esto puede ser la pantalla de un ordenador, teléfono inteligente o consola.
- **Elemento de procesamiento**, aquello que es capaz de interpretar la información del mundo real, para generar información virtual y unirlas de manera correcta, aquí nuevamente podemos hablar de ordenadores, teléfonos inteligentes y consolas.
- **Elemento activador de realidad aumentada**, aquí hablamos de elementos de localización como los GPS, brújulas y acelerómetros que permiten identificar la posición orientación de dichos dispositivos, así también como las etiquetas RFID o códigos bidimensionales, o en general cualquier elemento que es capaz de proporcionarnos lo que percibe el usuario.

3.2 Realidad aumentada y su impacto en el mundo

Aparentemente para algunos la realidad en sí misma no parece suficiente, los objetos físicos ofrecen a veces pocas oportunidades de interacción y por ello hay un número creciente de compañías que se interesan por la realidad aumentada (RA).

La tecnología RA es aplicada en muchos sectores, como la educación, medicina, entretenimiento, etc. Algunos ejemplos del uso de la realidad aumentada en algunas compañías son las siguientes.

- La empresa sueca IKEA, ha sido pionera en el uso de RA en sus catálogos, el cual te permite visualizar los muebles que te interesan dentro de una habitación.

- Volkswagen, introdujo un "sistema de asistencia técnica móvil con realidad aumentada", usada para ayudar en el ensamblaje del coche hibrido XL1.

- En 2017 aparece la aplicación BanKnotes del Banco Central Ruso, el cual permitía saber si estos billetes eran verdaderos o falsos, gracias a los nuevos diseños en billetes de 200 y 2000 rublos, los cuales al ser escaneados por la cámara de un móvil, se debe observar una animación del monumento correspondiente al del billete.

Ya en el 2010 el The New Media Consortium presentó el informe Horizon, donde habla sobre la relevancia que tendría esta tecnología para la docencia, el aprendizaje y la investigación creativa en dos a tres años.

La empresa americana destinada a la tecnología Gartner, ha publicado en el 2017 su top 10 estratégicas, de tendencias tecnológicas; Donde la tendencia numero 7 es la experiencia inmersiva (ilustraciones 9 y 10).

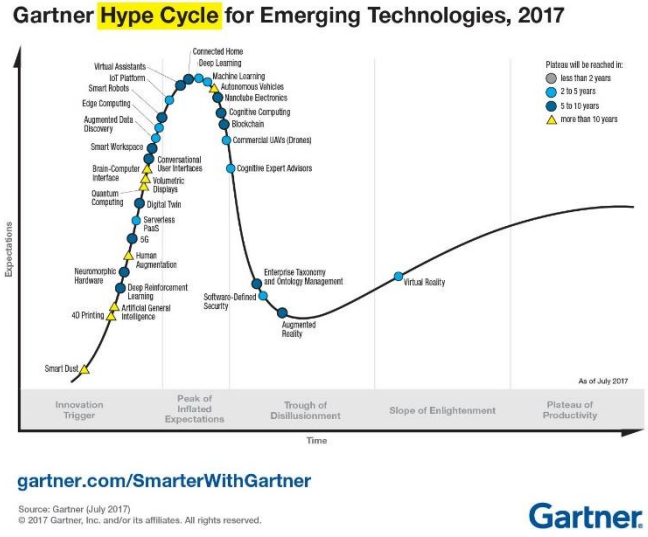

Ilustración 9. Las tecnológicas emergentes para 2018, según la empresa Gartner. [9]

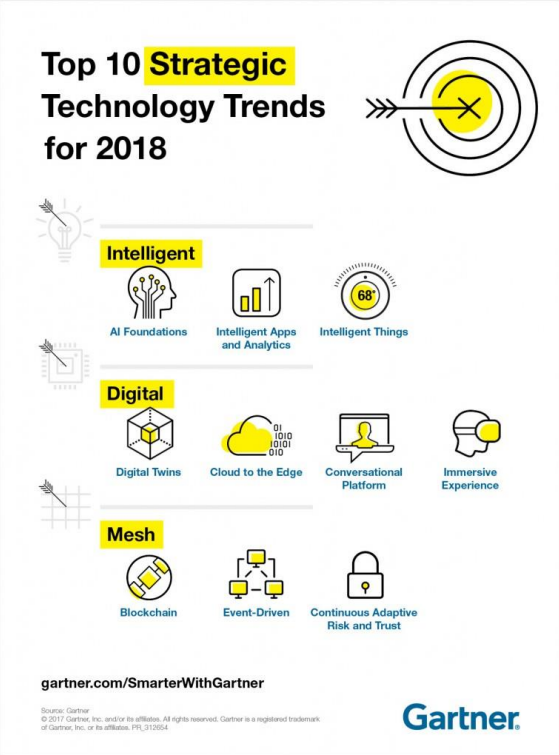

Ilustración 10. Las tecnológicas emergentes para 2018, según la empresa Gartner. [9]

Se puede observar en la ilustración 9, que la realidad aumentada está pasando por una etapa de desilusión, pero con la posibilidad de convertirse en una tecnología adoptada en unos 5 a 10 años futuros y comience a generar beneficios en las empresas.

3.3 HoloLens

La empresa Microsoft saca al mercado en el 2015, las gafas de realidad aumentada HoloLens. Este dispositivo nos permite crear un entorno más inmersivo, algo llamado realidad mixta.

La realidad mixta es el espectro existente entre la realidad física y la realidad digital (ilustración 11), se puede considerar como la mezcla entre la realidad, realidad aumentada, virtualidad aumentada, la realidad virtual y lo digital.

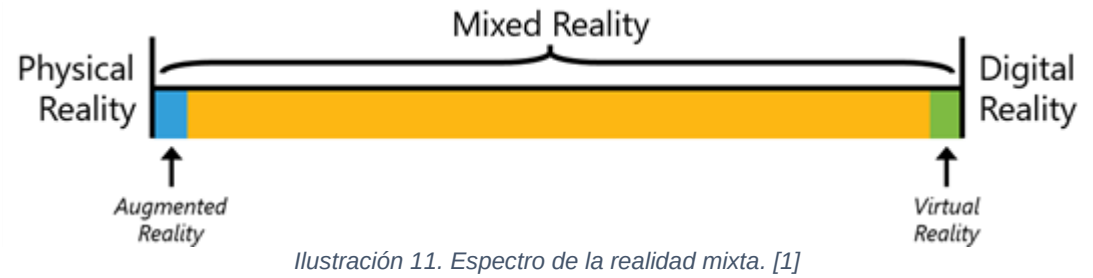

Ilustración 11. Espectro de la realidad mixta. [1]

Otra manera de comprender la realidad mixta, es según los dispositivos del que hacemos uso (ilustración 12).

- **Dispositivos holográficos**, estos se caracterizan por añadir contenido digital en la realidad como si realmente estuvieran ahí.
- **Dispositivos inmersivos**, estos se caracterizan por la capacidad de crear la sensación de "presencia", en un mundo virtual ocultando el mundo físico.

Ilustración 12. Tipos de dispositivos dentro del espectro de la realidad aumentada

HoloLens y su Windows 10, proporcionan una plataforma común de realidad mixta, para fabricantes y desarrolladores de dispositivos. Los dispositivos actuales permiten una posición o rango específico dentro del espectro de realidad mixta (ilustración 13), aunque con el tiempo deben expandir ese rango.

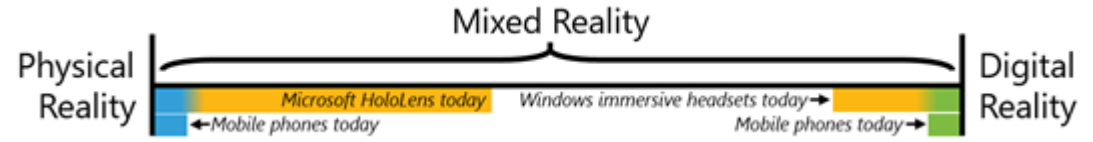

Ilustración 13. Dispositivos dentro del espectro de realidad aumentada

Aunque a menudo es mejor pensar en que tipo de experiencia queremos conseguir con nuestras aplicaciones dado que esto nos permitirá ubicarnos mejor en el espectro de realidad mixta (ilustración 14).

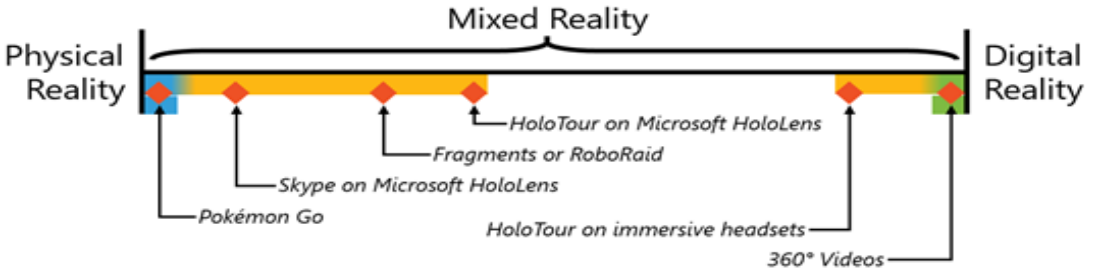

Ilustración 14. Aplicaciones dentro del espectro de la realidad mixta

3.4 ¿Qué es un holograma?

HoloLens permite crear y visualizar hologramas, que son objetos de luz y sonido que aparecen en el escenario que rodea al usuario como si fueran objetos reales; Estos objetos responden a diferentes tipos controles básicos de entrada, como son la mirada, un gesto o un comando de voz.

Los hologramas son objetos de luz, esto quiere decir que HoloLens no reproduce el contenido de color negro, aparecerá como transparente lo cual es conveniente usar, al buscar que nuestros objetos holográficos se superpongan a la realidad.

HoloLens reproduce los hologramas dentro una zona visual óptima, denominado marco holográfico.

3.4.1 Zona óptima para un holograma.

HoloLens aconseja como óptimo la distancia de dos metros (ilustración 15), dado que la experiencia empieza a degradarse cuanto más cerca de un metro esté.

Los hologramas que se mueven regularmente en profundidad tienen mayores probabilidades de causar problemas, que los hologramas estacionarios a la hora de ser reproducidos, por lo que hay que tener en consideración la de recortar o atenuar el contenido cuando el usuario se mueva por el escenario.

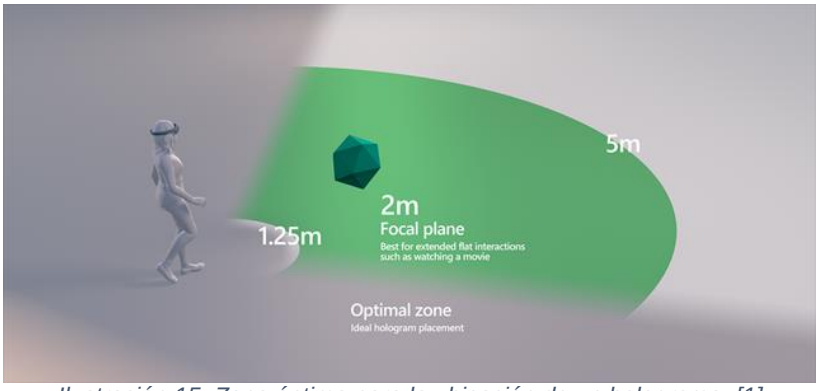

Ilustración 15. Zona óptima para la ubicación de un holograma. [1]

3.5 Controles básicos en HoloLens

Haremos mención a las entradas básicas que permite el dispositivo como la mirada y el gesto, los cuales hemos implementado para nuestra aplicación.

Se conoce la posibilidad que ofrece HoloLens para controlar hologramas mediante comandos de voz, pero es propio de un trabajo aparte dado que añadir palabras nuevas a su diccionario no es algo trivial.

3.5.1 Mirada

La mirada es la primera forma de entrada y es una forma primaria de focalización, le indica al usuario donde se está apuntando dentro de una escena.

La mirada es un vector y puedes pensar en este vector como un puntero láser (ilustración 16). A medida que el usuario mira alrededor de la escena, su aplicación puede interceptar este laser, tanto con sus propios hologramas como con una malla de mapeo espacial.

Aunque un punto a aclarar es que realmente lo que se usa para orientar el puntero láser, es la orientación de la cabeza del usuario y no la de sus ojos.

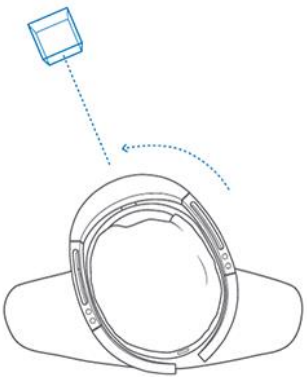

Ilustración 16. Mirada. [1]

En HoloLens, las interacciones generalmente deben derivar de la orientación de la mirada del usuario, en lugar de tratar de interactuar directamente con la mano.

La mayoría de aplicaciones ubican un cursor donde se encuentra el puntero láser para dar confianza al usuario con respecto al objeto con el que va a interactuar; Normalmente colocas este cursor en el escenario cuando el láser interactúa con un objeto, que puede ser un holograma.

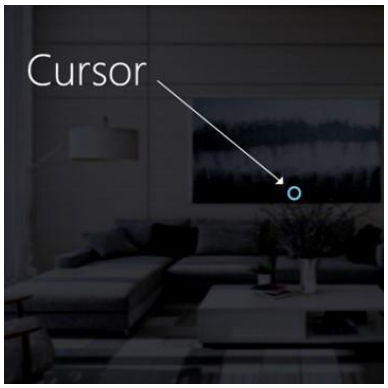

Ilustración 17. Ejemplo de un cursor en HoloLens. [1]

Dado que en el proyecto hemos hecho uso del emulador de HoloLens, mencionaremos como se simularía estas funciones dentro del emulador.

- **Mirar hacia arriba, abajo, izquierda o derecha**: Hacer clic y arrastra el mouse o usa las teclas de flecha en tu teclado.

3.5.2 Gestos

Los gestos con las manos permiten a los usuarios actuar dentro del escenario. La interacción se basa en la mirada para apuntar y el gesto para actuar sobre cualquier elemento que haya sido golpeado por la mirada.

Los dos gestos principales de HoloLens son *Air Tap* (golpe en el aire) y *Bloom* (florecer). Estas dos interacciones son el nivel más bajo de datos de entrada a los que puede acceder un desarrollador, forman la base de una variedad de posibles acciones de usuario.

Air Tap, es un gesto de tocar con la mano en posición vertical, similar a un clic del mouse o seleccionar. Esto se usa en la mayoría de las experiencias de

HoloLens. Para realizar esta acción primero levante su dedo índice a la posición de listo y como segundo paso presione dicho dedo hacia abajo para tocar el objeto mirado (ilustración 18).

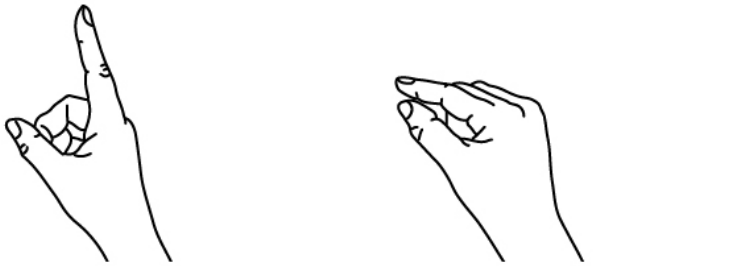

Ilustración 18. Gesto Air Tap. [1]

Bloom, es el gesto de *home* y está reservado solo para eso, es una acción especial del sistema que es equivalente a presionar la tecla Windows en un ordenador, el usuario puede usar cualquier mano. Para realizar esta acción debe tener la mano en vertical, juntar todas las yemas de los dedos y luego abrir la palma de la mano (ilustración 19).

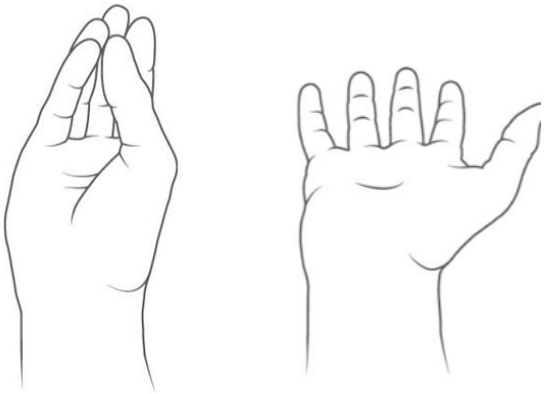

Ilustración 19. Gesto Bloom. [1]

Las aplicaciones pueden reconocer más que solo estos gestos y se pueden realizar gestos compuestos más complejos, estos gestos compuestos o de alto nivel se basan en los datos de entrada de bajo nivel anteriormente expuestos.

- ***Tap and hold***, es realizar la acción *Air Tap* pero manteniendo el dedo índice presionado cuando realice la segunda acción.
- ***Manipulation***, Los gestos de manipulación se pueden usar para mover, cambiar el tamaño o rotar un holograma cuando se desee. La orientación inicial para un gesto de manipulación debe hacerse mirando o señalando. Una vez se inicia el *Tap and Hold*, cualquier manipulación del objeto se maneja con movimientos manuales, lo que libera al usuario de mirar alrededor mientras manipulan.

- ***Navigation***, Los gestos de navegación funcionan como un joystick virtual y se pueden usar para navegar por los widgets de UI. Para esta acción de debe mantener la mirada en el objeto mientras se realiza el gesto *Tap and Hold* y se desplaza la mano.

En nuestra aplicación los gestos implementados son Air Tap y Manipulation.

HoloLens tiene sensores que pueden ver a pocos metros a cada lado del usuario. Cuando se usen estos gestos, se necesitara mantenerlos dentro de ese marco visión (ilustración 20) o no serán reconocidas. A medida que el usuario se mueva el marco se mueve con él.

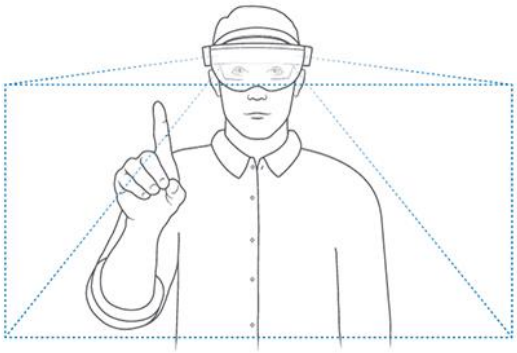

Ilustración 20. Vista frontal de un usuario de HoloLens. [1]

Dado que en el proyecto hemos hecho uso del emulador de HoloLens, mencionaremos como se simularía estas funciones dentro del emulador.

- **Gesto Air Tap:** Gesto manual equivalente a hacer clic con un mouse. Haga clic con el botón derecho del mouse, presione la tecla Enter en su teclado.
- **Gesto Bloom**: Gesto manual equivalente a presionar el teclado Home en el sistema operativo Windows. Presione la tecla Windows o la tecla F2 en su teclado.
- **Gesto Manipulation**: Gesto manual derivado del Air Tap que sirve para desplazar y rotar objetos virtuales. Mantenga presionada la tecla Alt mas el botón derecho del mouse y arrastre el mouse hacia arriba o abajo.

3.6 Tipo de aplicaciones soportados por HoloLens

Una de las ventajas al desarrollar aplicaciones para HoloLens, es que es un dispositivo que no te limita el tipo de entorno al cual está orientado tu aplicación.

3.6.1 Aplicaciones de entornos mejoradas

Este enfoque es de las más poderosas para aportar valor a los usuarios, se consigue facilitando la colocación de información o contenido digital en el entorno actual de un usuario (ilustración 21).

Este enfoque es popular para las aplicaciones donde la ubicación contextual del contenido digital en el mundo real es primordial. Este enfoque también permite a

los usuarios pasar fácilmente de las tareas del mundo real a las tareas digitales y viceversa.

Ilustración 21. Aplicación de entorno mejorado. [1]

3.6.2 Aplicaciones de entornos mezcladas

Dada la capacidad de HoloLens para crear una capa digital que puede superponerse por completo en el espacio del usuario. Debe respetar la forma y los límites del entorno del usuario, pero la aplicación puede optar por transformar ciertos elementos para así mejorar la inmersión del usuario a la aplicación (ilustración 22).

Ilustración 22. Aplicación de entorno mezclado. [1]

3.6.3 Aplicaciones de entorno inmersivo

Este tipo de aplicaciones se centran en un entorno que cambia por completo el mundo del usuario. Estos espacios pueden parecer reales, separando totalmente al usuario de la realidad y reemplazándolo por uno propio, esto es conocido como la realidad virtual. Aquí la inmersión solo está limitada por el tamaño del escenario de la aplicación (ilustración 23).

Ilustración 23. Aplicación de entorno inmersivo. [1]

3.7 HoloLens y su uso profesional

HoloLens tiene el objetivo de no ser solamente una herramienta de trabajo ni de usarlo sólo para una tarea específica.

Ya existen empresas que las utilizan, como la NASA con *Sidekick* que se usa para la capacitación de astronautas o el proyecto *OnSight* que permite a los usuarios caminar libremente por un espacio de demostración, Volvo que las usa para el diseño de sus coches y la empresa Zerintia Technologies experta en dar soluciones industriales basadas en nuevas tecnologías, donde algunas de sus soluciones usan HoloLens como plataforma.

CAPITULO 4. UNITY EN RED (UNET)

En este capítulo hacemos una descripción y mención de los diferentes componentes de la tecnología que ofrece Unity para el desarrollo de aplicaciones en red. Algunos de estos componentes descritos han sido aprovechados para el desarrollo de nuestra aplicación.

Antes de continuar con este capítulo, queremos marcar los siguientes conceptos utilizados.

- Cuando hacemos mención de la palabra jugadores, clientes o servidores realmente nos referimos a controladores aéreos según el rol que hayan asumido (local o externo).
- Cuando hacemos mención de la palabra juego, realmente nos referimos a la aplicación en red.
- Cuando hacemos mención de la palabra GameObjects en red, realmente nos referimos a los hologramas en forma de avión o aquellos que identifican a los controladores aéreos, los cuales se generan en nuestra aplicación.

4.1 Descripción general del multijugador

Hay dos maneras que un desarrollador puede utilizar las característica de red (*Networking*) a la hora de realizar un proyecto.

- **Los desarrolladores realizando un juego multijugador con Unity**; Estos desarrolladores deberían comenzar con un componente administrador de red (*NetworkManager*) o la API de Alto Nivel.
- **Los desarrolladores construyendo una infraestructura de red o juegos avanzados multijugador**; Estos desarrolladores deberían comenzar con la API usando el componente de transporte de red (*NetworkTransport*).

La API de Unity se puede ver como un modelo de capas (ilustración 24) el cual se va modificando o haciendo uso, dependiendo la necesidad del desarrollador.

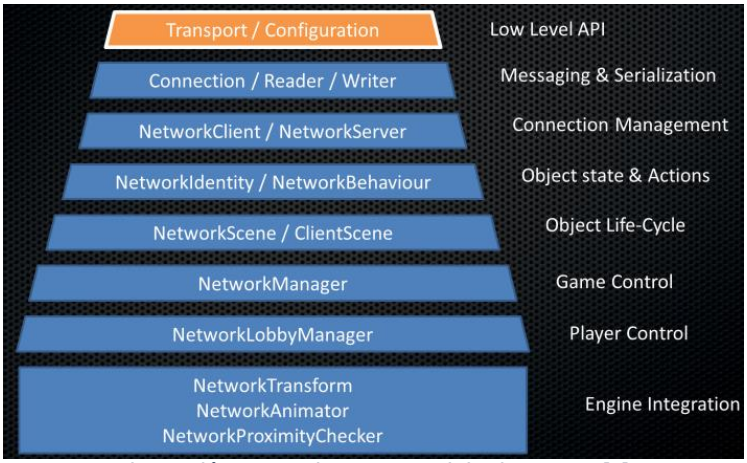

Ilustración 24. Api UNET, modelo de capas. [4]

4.1.1 API de scripting de alto nivel

Unity permite implementar la estructura de red, utilizando una API de alto nivel (*Hight level Api - HLAPI*). Esto significa que se tiene acceso a los comandos que cubren las mayorías de requerimientos de nuestra aplicación.

HLAPI de Unity proporciona.

- **Controlar el estado de red del juego utilizando un NetworkManager.**
- **Operar juegos alojados al cliente, donde el *host* es también un cliente**. Un controlador aéreo asume el rol de servidor, sin dejar de ser cliente.
- **Serializar datos utilizando un serializador con un propósito general**.
- **Enviar y recibir mensajes de red**. Útil para la interacción entre los diferentes controladores aéreos.
- **Enviar comandos de red de clientes a servidores**. Útil para las comunicaciones entre controladores aéreos externos hacia el local.
- **Realizar llamados de procedimientos remotos (*calls-RPCs*) de servidores a clientes**. Útil para la interacción entre el controlador aéreo local hacia los controladores aéreos externos.
- **Enviar eventos de red de servidores a clientes**.

4.1.2 Integración de motor y editor

La funcionalidad de red está integrado al motor y editor de Unity, permitiendo trabajar con componentes y ayudas visuales para construir un entorno multijugador. Lo que proporciona.

- Un componente de **NetworkIdentity** para GameObject de red.
- Un **NetworkBehaviour** para scripts en red.
- Una sincronización automática y configurable de los componentes **Transforms** que existen en los GameObject.
- Una sincronización automática de variables de script.
- Soporte para colocar objetos en red en escenas de Unity.
- Network componentes.

4.1.3 Capa de transporte en tiempo real de NetworkTransport

Unity incluye una capa de transporte red el cual ofrece.

- Un protocolo basado en UDP optimizado.
- Un diseño multicanal para evitar problemas de *head-of-line blockin* (*HOL*)
- Soporte para una variedad de servicios de calidad (*Queality of Service – QoS*) por canal.
- Una topología de red flexible que soporta arquitecturas de persona a persona o de cliente a servidor.

4.2 Conceptos sobre HLAPI

HLAPI es un sistema para construir juegos multijugador en Unity. Está construido por encima de la capa transporte.

La capa de transporte, soporta cualquier tipo de topología de red, permite la comunicación en tiempo real, y maneja las tareas comunes que son requeridas en una aplicación en red.

HLAPI es un conjunto de comandos de red, construidos dentro de un nuevo namespace **UnityEngine.Networking**, el cual proporciona lo siguientes servicios.

- Manejadores de mensajes.
- Serializadores de alto rendimiento de propósito general.
- Una administración de objetos distribuida.
- Sincronización de estados.
- Clases de red: *Server*, *Client*, *Conecction*, etc.

4.2.1 Sistema de red HLAPI

En el sistema HLAPI de Unity los juegos multijugador incluyen.

- **Servidor**, es una instancia del juego al que todos los demás jugadores se conectan.
- **Cliente**, son instancias del juego que generalmente se conectan desde diferentes dispositivos al servidor, los clientes pueden conectarse a través de una red local o en línea.

En cuanto al servidor, este puede ser un servidor dedicado o un servidor anfitrión también llamado *host* (ilustración 25).

- **Servidor dedicado**, esta es una instancia del juego que solo se ejecuta para actuar como servidor.
- **Servidor Anfitrión**, es un cliente que crea una única instancia del juego llamado host, que actúa como servidor y cliente.

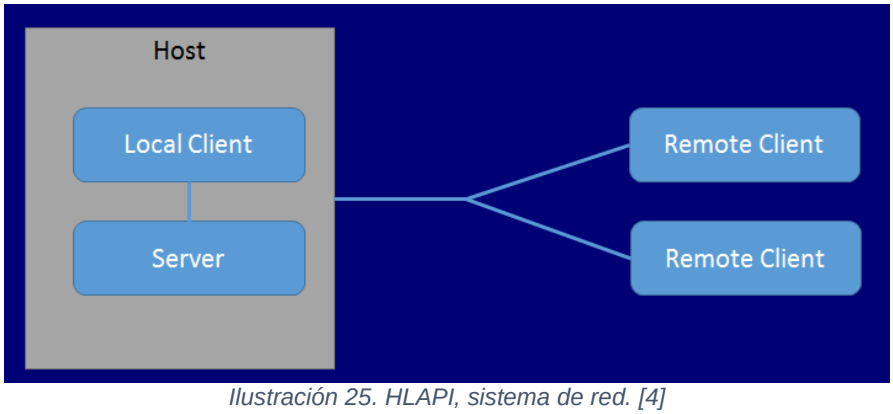

Ilustración 25. HLAPI, sistema de red. [4]

El *host* es una instancia única del juego, actúa como servidor y cliente; A este cliente se le denomina cliente local (*LocalClient*) mientras que a los otros clientes se les denomina clientes remotos (*RemoteClient*)

El cliente local se comunica al servidor a través de llamadas directas a funciones y colas de mensajes, ya que en realidad comparte la misma escena con el servidor.

El cliente remoto se comunica con el servidor sobre una conexión regular de red.

Con esta estructura se busca que el código para cliente local y remoto sea el mismo, de modo que solo se tenga que pensar en un tipo de cliente, cuando se esté desarrollando la aplicación.

4.2.2 GameObjects en red y su generación.

Cuando hablamos de un GameObject nos referimos a un objeto de los muchos que existen en un juego y **GameObject.Instantiate** es la herramienta que nos permite crear dichos objetos de manera local, lo cual no es del todo útil cuando hablamos de juegos multijugador, donde buscamos que estos objetos sean visibles y manipulables por otros clientes, para cubrir esta necesidad se usa los GameObject en red.

Los GameObjects en red, son objetos que están controlados y sincronizados por el sistema de red de Unity. Lo cual permite crear una experiencia compartida para todos los clientes, donde todos ven y escuchan lo mismos eventos o acciones, a pesar de que cada cliente tiene su propio punto de vista.

Es aquel objeto que tiene el componente **NetworkIdentity**, que permitirá al **NetworkManager** sincronizar su creación (*Spawning*) y su destrucción (*Destroy*),

Las propiedades que deben sincronizarse en cada GameObject en red, depende de la finalidad que se le quiere dar al objeto, estos son algunos ejemplos.

- La **posición** y **rotación**, en GameObject en movimientos, como los jugadores.
- El **estado de animación** de un GameObject que sea animado.
- El **valor** de una variable, como por ejemplo el tiempo que falta para terminar una partida.

4.2.3 El GameObject en red jugador

Cuando un cliente se une al servidor, el sistema HLAPI crea en el cliente un GameObject en red, al que se le asocia su conexión y que representa al cliente en el servidor.

HLAPI al hacer esto crea un sistema autoritativo, donde enruta los comandos de red (llamadas de procedimiento de cliente a servidor) a un solo GameObject, es decir que un jugador no puede invocar un comando en el GameObject de otro jugador.

La clase **NetworkBehaviour** tiene una propiedad denominada **isLocalPlayer**, el cual se establece como verdadero después de invocar a la función **OnStartLocalPlayer(),** el cual nos permite distinguir diferentes clientes dentro de una misma escena.

4.2.3 Autoridad de red

Los servidores y los clientes pueden administrar el comportamiento de los GameObject.

El estado de autoridad predeterminado en los juegos de Unity que utilizan HLAPI, es que el servidor tenga la autoridad sobre todos los GameObject que no representen jugadores, ya sean NPC, plataformas o cualquier objeto con el que el jugador pueda interactuar (ilustración 27).

Los jugadores también pueden tener autoridad sobre un GameObject que normalmente son GameObjects que sirven para identificar al propio jugador (ilustración 26).

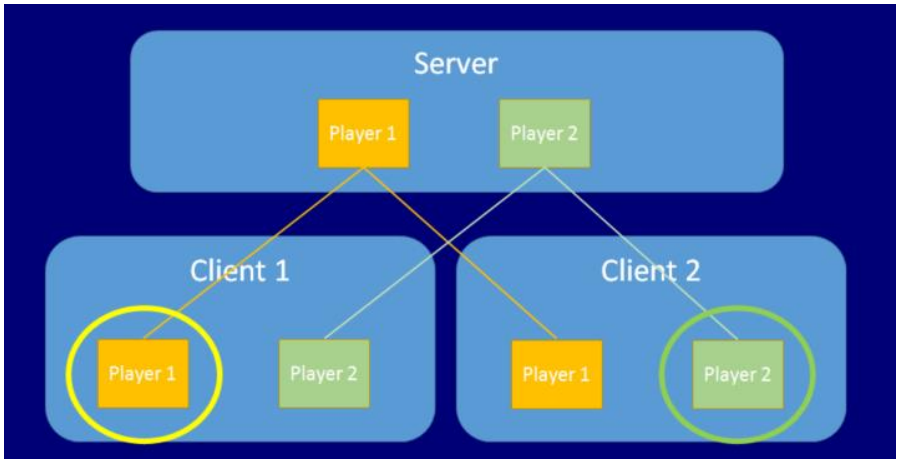

Ilustración 26. HLAPI, autoridad de red de un cliente sobre el GameObject que le identifica. [4]

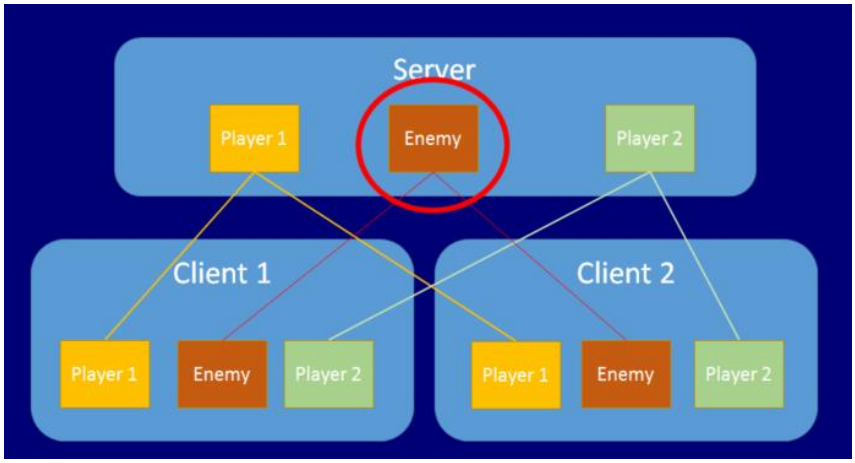

Ilustración 27. HLAPI, Autoridad de red de un servidor sobre un Gameobject del juego. [4]

4.2.4 Autoridad local para GameObject no jugadores.

Es posible tener autoridad local sobre los GameObjects que no son jugadores, y existen dos maneras de hacerlo.

- Usando la función **NetworkServer.SpawnWithClientAutority()**, para generar los GameObject, donde se pasa como parámetro la conexión de red del cliente.
- Usando la función **NetworkIdentity.AssignClientAuthority()**, donde se pasa como parámetro la conexión de red del cliente para tomar posesión de un GameObject ya existente.

Cuando se genera un GameObject donde la autoridad la tenga un cliente, se llama a la función **OnStartAuthority()** que devolverá la propiedad **hasAuthority** como verdadero; Tanto el método como la propiedad extienden de la clase **NetworkBehaviours.**

4.2.5 Sincronización del estado de variables

La sincronización de estados se hace por parte del servidor hacia los clientes remotos, esta actualización se envía de manera automática cuando el **SyncVar** cambia su valor.

Los SyncVar pueden ser de tipos básicos como el int, strings y floats o más complejos como Vector3 y structs.

Una manera de aprovechar que la sincronización es automática, es relacionarlos con ganchos (*Hook*), que son funciones que se ejecutan cada vez que el SyncVar cambia de valor.

4.2.6 Acciones y comunicación

Unet tiene maneras de realizar acciones a través de la red. Este tipo de acciones se les denomina llamadas de proceso remoto (*Remote Procedure Calls - RPC*) y existen dos tipos de RPC (ilustración 28).

- Los **Commands** (comandos), son enviadas por los GameObjects jugadores en el cliente hacia su homólogo existente en el servidor, por seguridad, Commands solamente pueden enviarse desde este tipo de GameObject, evitando que un cliente tenga control sobre los otros. Las funciones Commands debe llevar el prefijo Cmd y el atributo personalizado **[Command].**
- Los **ClientRpc**, son enviadas por los GameObjects generados por el servidor o aquellos donde el servidor tiene autoridad, hacia su homólogo en los clientes remotos. Las funciones ClientRpc deben llevar el prefijo Rpc y el atributo personalizado **[ClientRpc]**.

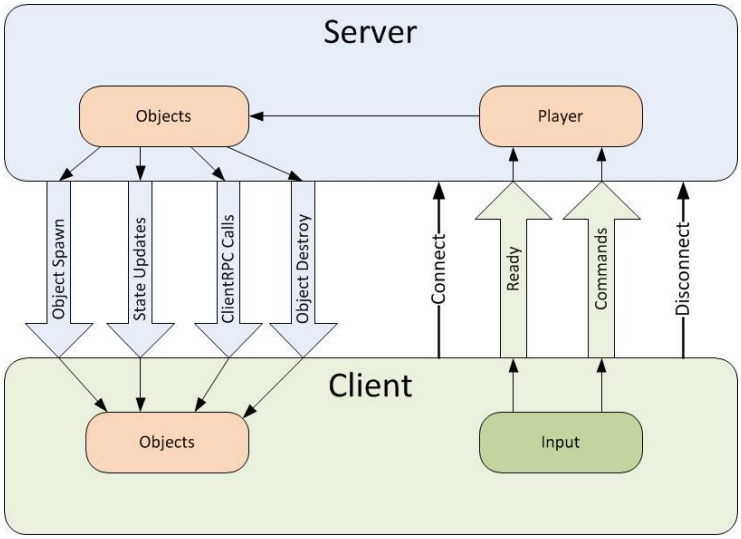

Ilustración 28. HLAPI, acciones remotas entre cliente y servidor. [4]

4.2.7 Visibilidad personalizada

Unity es compatible con la idea de que no todos los objetos deban ser visibles por todos los jugadores en el juego. A esto le podemos llamar área de interés, ya que los jugadores solo reciben visibilidad de los objetos que son relevantes o interesantes para ellos.

Esto es importante cuando hablamos de escenarios grandes donde el servidor contiene muchos objetos en red, lo cual la idea de reducir este número de objetos visibles es beneficioso para los jugadores, ya que reduce los tiempos de inicio de sesión y el ancho de banda en curso.

Hay maneras sencillas de restringir la visibilidad de los objetos, una es el uso del componente **NetworkProximityChecker** la cual te puede se implementa utilizando una interfaz de visibilidad publica Networking, usando esta interfaz, los desarrolladores son capaces de implementar cualquier tipo de reglas de visibilidad.

CAPITULO 5. IMPLEMENTACIÓN

En este capítulo, hacemos referencia a los prefabricados que se utilizan en la aplicación, así como los componente y scripts más relevantes para nuestra aplicación.

También tendremos un punto de vista a nivel de lenguaje de programación, del código puramente modificado por nuestra parte y no de los scripts importados.

5.1 Implementando HoloToolkit Inputs.

Unity permite importar a nuestra aplicación este paquete que ofrece Windows, lo que permite manejar varios tipos de entrada y enviarlos a cualquier objeto del juego que se esté mirando en cada momento, o cualquier objeto alternativo.

Cada fuente de entrada (manos, gestos, otros) implementa una interfaz **IInputSource**. La interfaz define varios eventos que las fuentes de entrada pueden activar.

Las fuentes de entrada se registran con **InputManager**, cuya función es reenviar las entradas a los objetos apropiados del juego. Las fuentes de entrada se pueden habilitar/deshabilitar dinámicamente según sea necesario, y se pueden crear nuevas fuentes de entrada para admitir diferentes dispositivos de entrada.

Aquí haremos mención a las fuentes de entrada más relevantes para la construcción de nuestra aplicación (en el anexo se podrá observar un diagrama con todas fuentes de entrada posibles).

En la aplicación se hace uso de un **InputManager** el cual escucha los diversos eventos provenientes de las fuentes de entrada, y también tiene en cuenta la mirada (*Gaze*). Actualmente, esa mirada siempre proviene de la clase **GazeManager**.

Los objetos de juego que desean añadir eventos de entrada, pueden implementar una o varias interfaces de entrada.

- **IInputHandler** para la fuente arriba y abajo. La fuente puede ser una mano que tocó, un clic que se presionó, etc.
- **IManipulationHandler** para el gesto de manipulación de Windows.

HoloToolkit, contiene prefabricados (GameObject con componentes y scripts añadidos), que ya tienen relación con las funciones de entrada, haremos mención de los prefabricados utilizados y de algunos de sus scripts más destacables, sin entrar al detalle dado que las usamos como herramientas para agilizar el desarrollo de nuestra aplicación y en ningún momento buscamos personalizar o reconstruir estas herramientas.

5.1.1 InputManager.prefab

Sistema de entrada que maneja la mirada y varias fuentes de entrada compatibles con HoloLens, como las manos y los gestos (ilustración 34).

Esto también incluye una fuente de entrada falsa que le permite simular la entrada cuando está en el editor de Unity. Por defecto, esto se puede hacer manteniendo presionadas la tecla *Shift* (fuente de entrada izquierda) o *Space* (fuente de entrada derecha), moviendo el mouse para mover la fuente y usando el botón izquierdo del mouse para tocar.

InputManager.cs

Componente Singlenton a cargo de gestionar la clase InputManager, Input Manager es responsable de administrar las fuentes de entrada y despachar eventos relevantes

GazeManager.cs

Componente Singlenton a cargo de gestionar la clase GazeManger, aquí es donde se puede definir que capas son considerables al observar un objeto.

Opcionalmente el GazeManager puede hacer referencia a un estabilizador de mirada.

- **MaxGazeCollisionDistance**: la distancia máxima del *raycast* (emisión de rayo). Cualquier holograma más allá de este valor no se golpea.
- **RaycastLayers**: las capas de Unity contra *raycast*. Si tiene hologramas que no deberían golpear, como un cursor, no incluya sus capas en esta máscara.
- **Stabilizer**: Para estabilizar la mirada. Si no se indica, la mirada no se estabilizará.
- **GazeTransform**: la transformación para usar como fuente de la mirada. Si no está configurado, se establecerá de manera predeterminada en la cámara principal.

GazeStabilizer.cs

Estabilice la mirada del usuario para dar cuenta de la inestabilidad de la cabeza, extiende de la clase abstracta BaseRayStabilizer

- **StoredStabilitySamples** Número de muestras en las que desea iterar. Un número mayor será más estable.

BaseRayStabilizer.cs

Una clase abstracta que debe implementar alguna clase en concreto, para crear un estabilizador, que toma la posición de entrada y rotación, y realiza operaciones sobre ellos para estabilizar o suavizar esos datos.

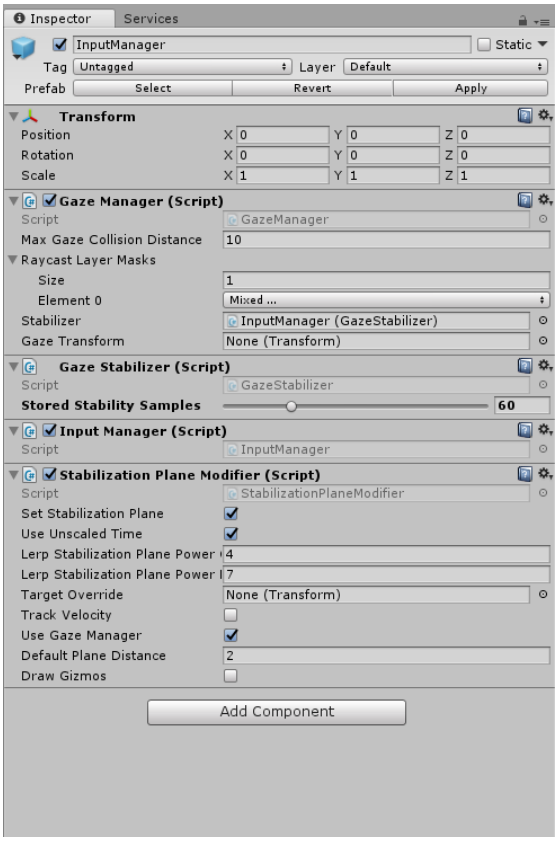

Ilustración 29. InputManager.prefab.

5.1.2 HoloLensCamera.prefab

La Cámara de Unity, se ha personalizado para el desarrollo holográfico (ilustración 35).

- Componente **Transform** establecido en 0,0,0
- Componente **Camera**, 'Clear Flags' cambió a 'Solid Color'
- El color se establece en R: 0, G: 0, B: 0, A: 0, ya que el negro se vuelve transparente en HoloLens.
- Establezca el plano de recorte cercano recomendado.
- Permite el movimiento manual de la cámara cuando está en el editor

ManualGazeControl.cs

Ayuda a la manipulación de la cámara dentro del editor de Unity, el usuario puede moverse por la escena usando las teclas WASD y mirar alrededor usando el botón derecho del mouse.

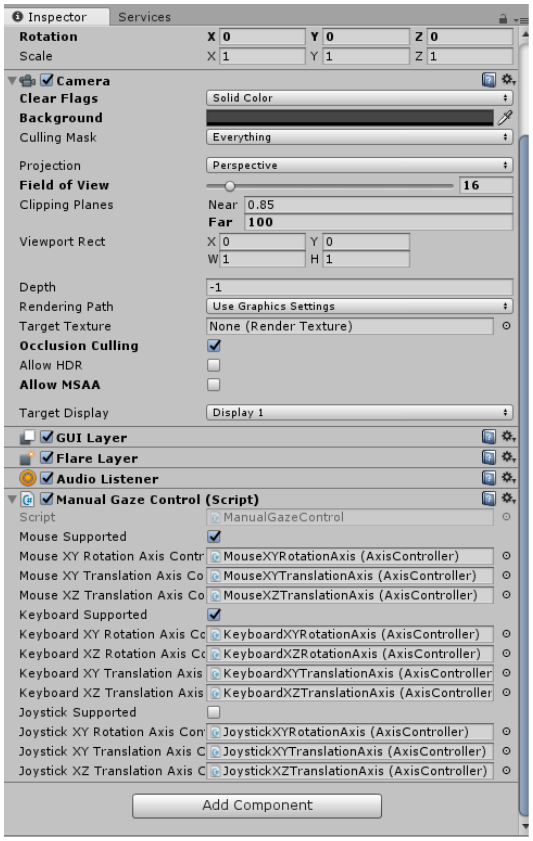

Ilustración 30. HoloLensCamera.prefab

5.1.3 Cursor.prefab

Toma forma según el estado del prefabricado; De doble anillo cuando el controlador aéreo esté apuntando hologramas, el estado pasa a CursorOnHolograms y un haz de luz cuando el usuario está mirando fuera de los hologramas, el estado pasa a CursorOffHolograms, este contiene el script ObjectCursor.cs (ilustración 36).

ObjectCursor.cs

Cursor cuyos estados están representados por uno o varios prefabricados, extiende de la clase abstracta Cursor.

Cursor.cs

Clase abstracta que debe implementar alguna clase en concreto, para facilitar la creación de un cursor personalizado. Esto proporciona la lógica básica para mostrar un cursor en la ubicación que está mirando el usuario.

- Decide cuándo mostrar el cursor.
- Posiciona el cursor en la mirada cuando un objeto es golpeado.
- Gira el cursor para que coincida con las normales del holograma.

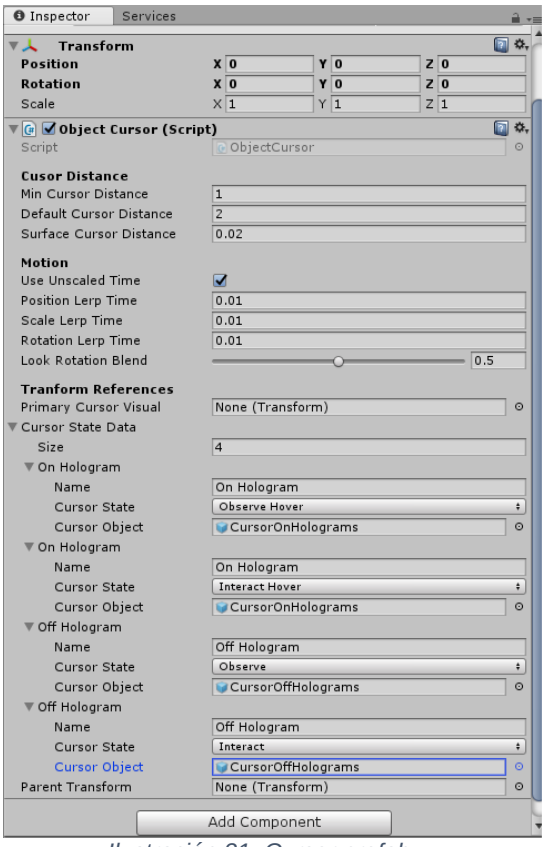

Ilustración 31. Cursor.prefab

5.2 Implementando HLAPI de Unity.

Para la aplicación se creó los siguientes prefabricados, los cuales aprovechan los componentes que ofrece la API de alto nivel de Unity, para el desarrollar aplicaciones en red.

5.2.1 UnetSharingStage.prefab

Prefabricado que hace uso de las clases NetworkManager y NetworkDiscovery, lo cual facilita la gestión de la aplicación, el engendrado de aviones, gestión de escenas, elección de modo de inicio de la sesión y del descubrimiento de sesiones (ilustración 32).

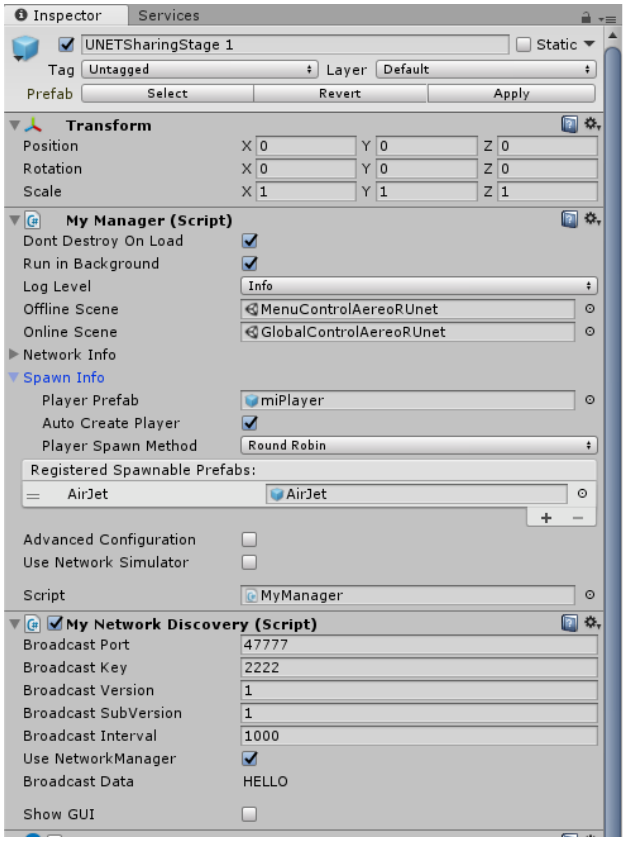

Ilustración 32. UnetSharingStage.prefab

5.2.2 MiPlayer.prefab

Prefabricado configurado, para que un GameObject se genere automáticamente por cada conexión nueva en el servidor. Esto se aplica al controlador aereo local que asume el rol de servidor (**host**) y a los controladores aéreos externos. Este prefabricado contiene los componentes NetworkIdentity, NetworkTransform y un script MyplayerController.cs (ilustración 33).

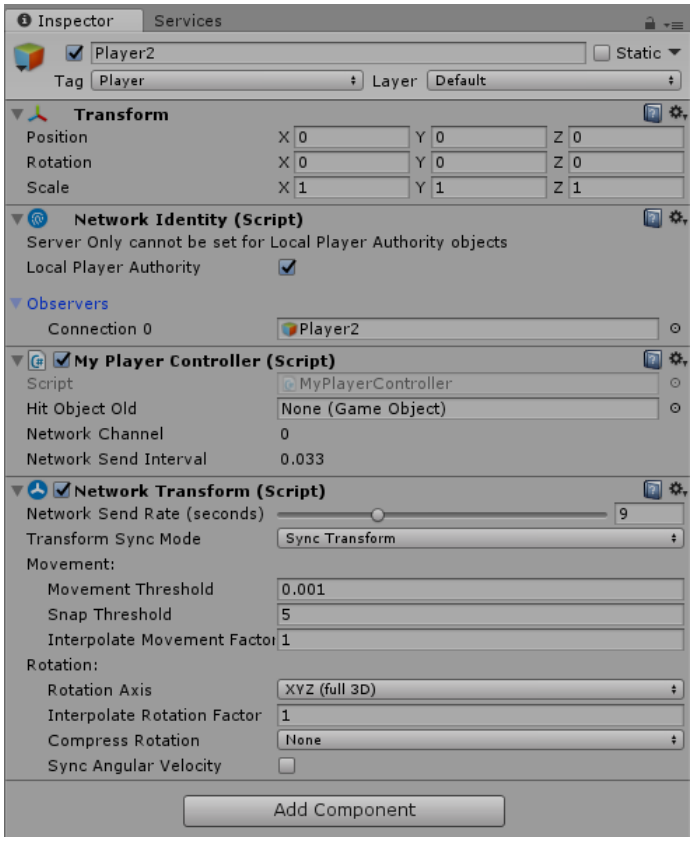

Ilustración 33. Miplayer.prefab

5.2.3 AirJet.prefab

Prefabricado configurado, para que los aviones se creen en base a este prefabricado cada vez que un controlador aéreo lo solicita. Este prefabricado contiene los componentes Rigibody, NetworkIdentity, NetworkTransform y un script PublicAvionController.cs (ilustración 34).

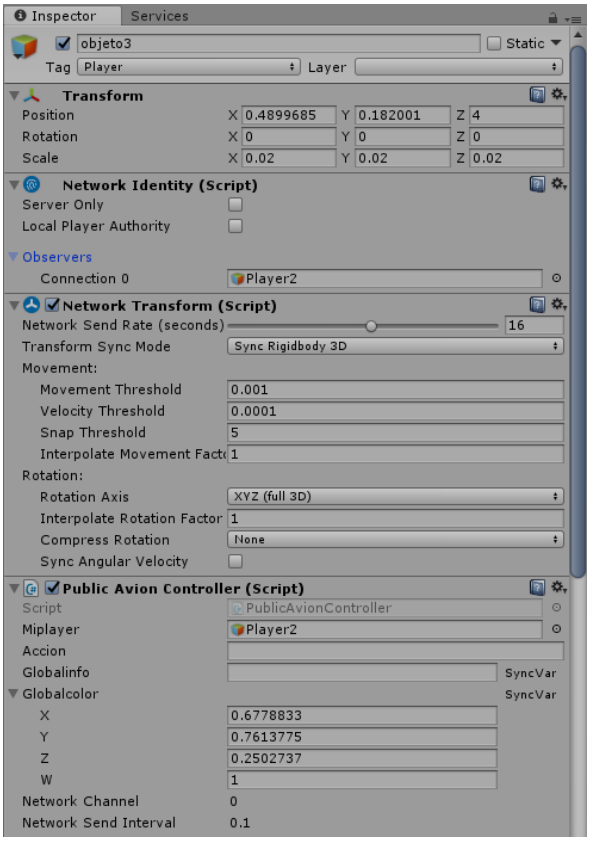

Ilustración 34. AirJet.prefab

5.2.3 Spawner.prefab

Prefabricado configurado, para ser el generador de aviones basados en el AirJet.prefab, estos se generan a petición de los controladores aéreos.

Este prefabricado solo se creara en la escena del servidor y contiene los componentes Rigibody, NetworkIdentity y un script AvionSpawner.cs (ilustración 35).

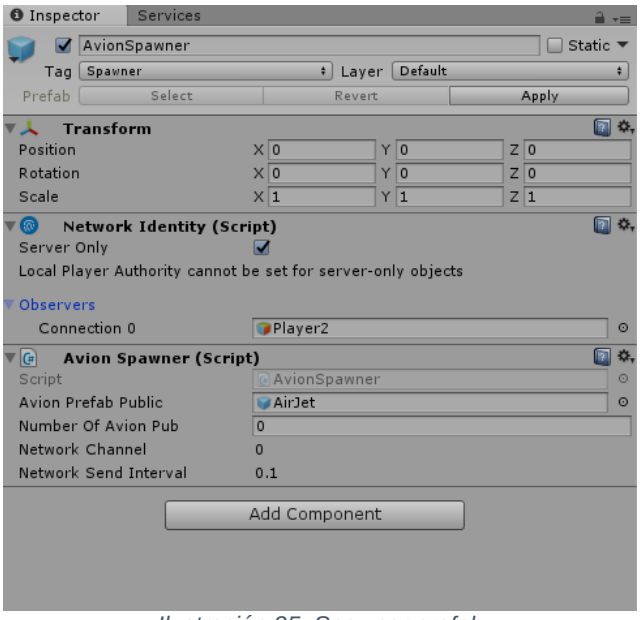

Ilustración 35. Spawner.prefab

5.3 Aplicación Implementada.

Aquí haremos mención a nivel de lenguaje de programación las acciones que se pueden realizar una vez iniciada, para esto hacemos referencia a los prefabricados anteriormente mencionados en los capítulos 5.1 y 5.2.

5.3.1 Cursor

El cursor está compuesto por dos prefabricados, denominamos hijos; Así pues el cursor cambiara el aspecto, dependiendo del prefabricado hijo que este activo.

Referencia al script ObjectCursor.cs del anexo.

```
public override void OnCursorStateChange(CursorStateEnum state)
 {
     base.OnCursorStateChange(state);
     if (state != CursorStateEnum.Contextual)
     {
         for(int i = 0; i < ParentTransform.childCount; i++)
         {
             ParentTransform.GetChild(i).gameObject.SetActive(false);
         }

         for (int i = 0; i < CursorStateData.Length; i++)
         {
             if (CursorStateData[i].CursorState == state)
             {
                 CursorStateData[i].CursorObject.SetActive(true);
             }
         }
     }
 }
}
}
```

5.3.2 Crear o unirse a la sesión

Una vez inicia la aplicación, el controlador aéreo puede decidir la manera en la que iniciara la sesión. Este puede decidir iniciar la sesión como controlador aéreo local (servidor) o como un controlador aéreo externo (cliente).

Referencia al script MainMenu.cs del anexo.

```
public void CreateGame () {
    if (myNetworkDiscovery.isClient)
    {
        return;
    }
    ...
    if (myNetworkDiscovery.StartAsServer())
    {
        NetworkManager.singleton.StartHost();
    }
}

public void SearchMatch() {
    if (myNetworkDiscovery.isServer){
        return;
    }
    ...
    if (!myNetworkDiscovery.StartAsClient())
```

```
    {
        myNetworkDiscovery.StartAsClient();
    }
}
```

MyNetworkDiscovery es un script que se encuentra en el prefabricado UnetSharingStage.prefab y que dicho script extiende de la clase NetworkDiscovery el cual nos permite iniciar como cliente o servidor, y en el caso de ser posible iniciar como servidor, también iniciamos como host.

Cuando un controlador aéreo asume el rol como servidor, empieza a difundir su conexión, mientras que los controladores aéreos externos empiezan a recibirla.

La función OnReceivedBroadcast de la clase NetworkDiscovery permite manejar los mensajes de difusión cuando su rol es el de controlador aéreo externo.

Referencia al script MynetworkDiscovery.cs del anexo.

```
public override void OnReceivedBroadcast(string fromAddress, string
data)
{
    if (!myDictionary.ContainsValue(fromAddress))
    {
      ...
        myDictionary.Add(index, fromAddress);
        flagmyDictionary = true; //para controlar el dropdown
        index++;
    }
}
}
```

En este punto vamos añadiendo a un diccionario todas las conexiones que hemos recibido, pero aunque nuestro rol sea de controlador aéreo externo, realmente no estamos asignados a una sesión.

Una vez elegido la dirección IP de la sesión que nos interesa, decidimos empezar como controlador externo (cliente) para ese controlador aéreo local (servidor).

Referencia al script MenuClient.cs del anexo.

```
public void JoinGame()
{
    //unirse a la partida
    ...
    string myopcion = myNetworkDiscovery.myDictionary[miList.value];
    ...
    NetworkManager.singleton.networkAddress = myopcion;
    NetworkManager.singleton.StartClient();
}
}
```

Tanto el controlador local (servidor) como el controlador aéreo externo (cliente), tendrán una representación de su conexión con la forma del prefabricado MyPlayer.prefab, esta representación se crea de manera automática gracias al componente NetworkManager del UnetSharingStage.prefab.

5.3.3 Crear Aviones

Una vez ya dentro del juego, tenemos la posibilidad de crear aviones indistintamente de nuestro rol como controladores aereos.

Referencia al script MenuSession.cs del anexo.

```
public void Create()
{
...
miplayer.GetComponent<MyPlayerController>().CmdCreate();
}
```

Como observamos instanciamos el prefabricado que representa nuestra conexión, y llamamos al método CmdCreate.

Referencia al script MyplayerController.cs del anexo.

```
[Command]
public void CmdCreate()
{
Debug.Log("MyPlayerController.cs-CmdCreate");
GameObject spawner = GameObject.FindWithTag("Spawner");
spawner.GetComponent<AvionSpawner>().OnCreateObject(connectionToClient
);
}
```

Este comando se ejecuta en el prefabricado homologo que existe en el servidor, y lo cual nos permite encontrar el prefabricado Spawner.prefab que solo existe en el cliente servidor, el cual finalmente creara el GameObject en red.

Referencia al script AvionSpawner.cs del anexo.

```
public void OnCreateObject(NetworkConnection firstobserver)
{
Debug.Log("AvionSpawner.cs-OnCreateObject");
var spawnPosition = new Vector3(
        Random.Range(0.0f, 0.5f),
        Random.Range(0.0f, 0.5f),
        4.0f
        );

var spawnRotation = Quaternion.Euler(0.0f,0.0f,0.0f);

var avionGlobal = (GameObject)Instantiate(avionPrefabPublic,
spawnPosition, spawnRotation);
avionGlobal.GetComponentInChildren<PublicAvionController>().globalcolo
r.Set(Random.value, Random.value, Random.value, 1.0f);//generamos un
color aleatorio
avionGlobal.GetComponentInChildren<PublicAvionController>().firstconec
tion = firstobserver;
NetworkServer.Spawn(avionGlobal);//crea el objeto
avionGlobal.name = "objeto" +
avionGlobal.GetComponent<NetworkIdentity>().netId.ToString();
avionGlobal.GetComponentInParent<NetworkIdentity>().RebuildObservers(t
rue);
}
}
```

Spawner.prefabs crea el GameObject en red lo hará basándose en el prefabricado AirJet.prefab y se le asigna la conexión (NetworkingConnection) de quien solicito su creación.

Este GameObject en red, empezara a recibir conexiones pero no se les permitirá ser observadores.

Referencia al script PublicAvionController.cs del anexo.

```
[Server]
public override bool OnCheckObserver(NetworkConnection newObserver)
{
if (newObserver.Equals(firstconection))
{
    return true;
}
return false;
}

[Server]
public override bool OnRebuildObservers(HashSet<NetworkConnection>
observers, bool initialize)
{
Debug.Log("PublicAvionController.cs-OnRebuildObservers");
if (initialize)//consideramos que reconstruye por primera vez
{
    for (int i = 0; i < NetworkServer.connections.Count; i++)
    {
      if(OnCheckObserver(NetworkServer.connections[i]))
        {
            observers.Add(NetworkServer.connections[i]);
        }
    }
    return true;
}
... }
```

5.3.4 Compartir o Soltar Aviones

Una vez se ha creado un avión (GameObject en red), este objeto tiene asignado unos observadores, el cual puede ser reconstruido por parte del controlador aéreo local (servidor) si un controlador aéreo externo lo solicita.

El controlador aéreo externo debe elegir el avión, e informar del identificador único de dicho objeto. Para que un controlador aéreo pueda elegir un avión, hemos decidido hacer uso del tipo de gesto Air Tap.

Referencia al script MyplayerController.cs del anexo.

```
public void OnInputClicked(InputClickedEventData eventData)
{
Debug.Log ("MyPlayerController.cs-OnInputClicked");
...
GameObject hitObject = GazeManager.Instance.HitObject;
if (hitObject.GetComponentInParent<NetworkIdentity>() != null)
{
hitObjectOld = hitObject;
...
}
return;}
```

Una vez elegido el avión, se podrá obtener su identificador único de red dado que usa el componente NetworkIdentity y que el NetworkServer le asigna un valor en el momento de su creación.

En este punto el controlador aéreo externo puede elegir si su prefabricado homólogo en el controlador aéreo local (servidor) debe compartir (Share) o soltar (Drop) el avión elegido.

Referencia al script MenuSession.cs del anexo.

```
public void Share()
{
...
GameObject obj =
miplayer.GetComponent<MyPlayerController>().hitObjectOld;
miplayer.GetComponent<MyPlayerController>().CmdShare(obj.GetComponentI
nParent<NetworkIdentity>().netId);
}
public void Drop()
{
...
GameObject obj =
miplayer.GetComponent<MyPlayerController>().hitObjectOld;
miplayer.GetComponent<MyPlayerController>().CmdDrop(obj.GetComponentIn
Parent<NetworkIdentity>().netId);
}
```

NetworkServer es capaz de encontrar cualquier avión, a través de su identificador único (netID).

Referencia al script MyplayerController.cs del anexo.

```
[Command]
public void CmdShare(NetworkInstanceId obj)
{
GameObject obj_server = NetworkServer.FindLocalObject(obj);
if (obj_server.GetComponentInParent<PublicAvionController>() != null)
{
...
obj_server.GetComponentInParent<PublicAvionController>().ServerShare(c
onnectionToClient);
}
return;
}
...
[Command]
public void CmdDrop(NetworkInstanceId obj)
{
GameObject obj_server = NetworkServer.FindLocalObject(obj);
if (obj_server.GetComponentInParent<PublicAvionController>() != null)
{
...
obj_server.GetComponentInParent<PublicAvionController>().ServerDrop(co
nnectionToClient);
}
return;
}
```

Una vez encontrado el avión, el servidor se encarga de reconstruir los observadores, añadiendo o eliminando observadores a través de las conexiones activas que tiene.

Referencia al script PublicAvionController.cs del anexo.

```
[Server]
public void ServerShare(NetworkConnection shareobserver)
{
conectionsharing = shareobserver;
accion = "SHARE";
GetComponentInParent<NetworkIdentity>().RebuildObservers(false);
}
[Server]
public void ServerDrop(NetworkConnection dropobserver)
{
conectiondroping = dropobserver;
accion = "DROP";
GetComponentInParent<NetworkIdentity>().RebuildObservers(false);
}
...
[Server]
public override bool OnRebuildObservers(HashSet<NetworkConnection>
observers, bool initialize)
{
...
if (!initialize)//consideramos que reconstruye otras veces
{

    if (accion.Equals("SHARE"))
    {
        for (int i = 0; i < NetworkServer.connections.Count; i++)
        {
            playersObserving.Add(NetworkServer.connections[i]);
            observers.Add(NetworkServer.connections[i]);
        }
    }
    else if (accion.Equals("DROP"))
    {
        if (playersObserving.Remove(conectiondroping))
        {
           foreach(NetworkConnection obs in playersObserving)
            {
                observers.Add(obs);
            }
        }
    }
    return true;
}
return false;
}
```

5.3.5 Modificar Aviones

Una vez creado el avión, este tiene variables asignadas que pueden ser modificadas por los controladores aéreos y que dichas modificaciones sean visible por todos los observadores.

El controlador aéreo debe elegir el avión, e informar del identificador único de dicho objeto. Para que un controlador aéreo pueda elegir un avión, hemos decidido hacer uso del tipo de entrada Air Tap.

Referencia al script MyplayerController.cs del anexo.

```
public void OnInputClicked(InputClickedEventData eventData)
{
...
GameObject hitObject = GazeManager.Instance.HitObject;
if (hitObject.GetComponentInParent<NetworkIdentity>() != null)
{
...
NetworkInstanceId netID =
hitObject.GetComponentInParent<NetworkIdentity>().netId;
CmdMove(netID);
}
return;
}
...
[Command]
void CmdMove(NetworkInstanceId obj)
{
GameObject obj_server = NetworkServer.FindLocalObject(obj);
...
if (obj_server.GetComponentInParent<PublicAvionController>() != null)
{
obj_server.GetComponentInParent<PublicAvionController>().ServerMove();
}
}
```

Para este ejemplo usaremos una variable SyncVar del tipo variable Vector4, el cual determinara el color del avion. Este SyncVar tiene un gancho (hook) que se ejecuta cuando a dicho SyncVar se le modifica el valor.

Referencia al script PublicAvionController.cs del anexo.

```
[SyncVar]
public string globalinfo = "";

[SyncVar(hook = "OnChangeColor")]
public Vector4 globalcolor;
...

[Server]
public void ServerMove()
{
...
ServerUpdateInfo();
transform.Translate(Vector3.left * 0.0f);
GetComponentInChildren<MeshRenderer>().material.color = globalcolor;
}
...
[Server]
void ServerUpdateInfo()
{
...
Vector4 localcolor = new Vector4();
localcolor.Set(Random.value, Random.value, Random.value, 1.0f);
globalcolor = localcolor;
}
```

```
...
public void OnChangeColor(Vector4 color)
{
GetComponentInChildren<MeshRenderer>().material.color = color;
}
```

CAPITULO 6. RESULTADOS

En este capítulo, ilustraremos los resultados obtenidos en base a la aplicación desarrollada para este proyecto final de grado. Esta aplicación está sigue la estructura lógica mostrada en la ilustración 8. Para una mejor visualización de los resultados se hará uso del componente NetworkManagerHUD.

Aunque la aplicación sea para una red local, el emulador HoloLens debe tener conexión a internet, para su correcta configuración consulte el anexo.

6.1 Iniciando aplicación.

Inicialmente ejecutamos la aplicación tanto en el entorno de Unity3D como en el emulador de HoloLens. En las ilustraciones 36 y 37 observamos que ambos tienen la posibilidad de iniciar la sesión o unirse a alguna.

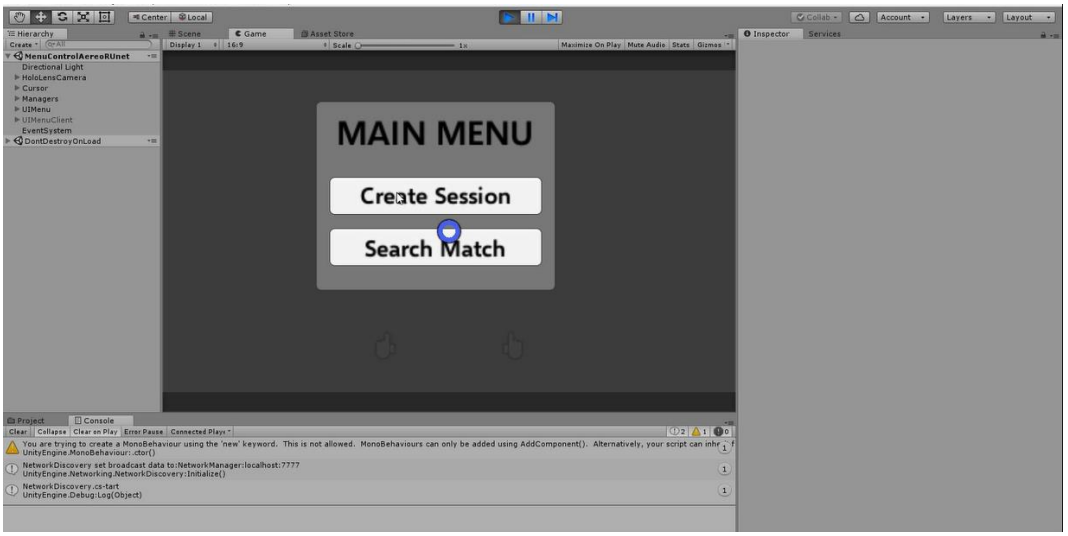

Ilustración 36. Unity3D, inicio de aplicación

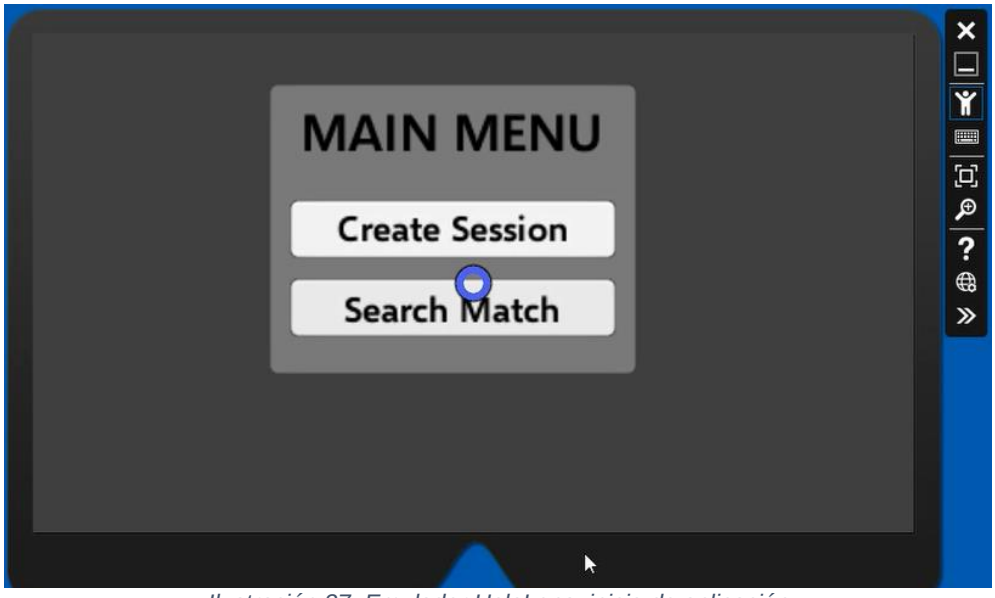

Ilustración 37. Emulador HoloLens, inicio de aplicación

6.2 Crear o unirse a una partida

En este caso, la aplicación iniciada en Unity3D será quien asuma el rol de controlador aéreo local como se puede observar en la ilustración 38.

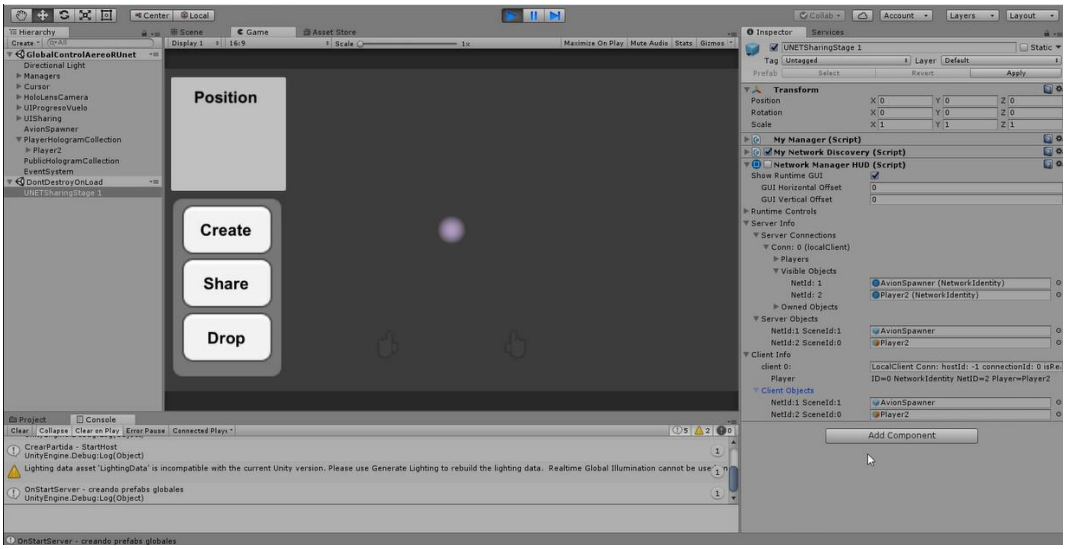

Ilustración 38. Unity3D, sesión iniciada

En la ilustración 39 podemos observar el componente NetworkManagerHUD, el cual nos muestra que el controlador aéreo local solo tiene una conexión.

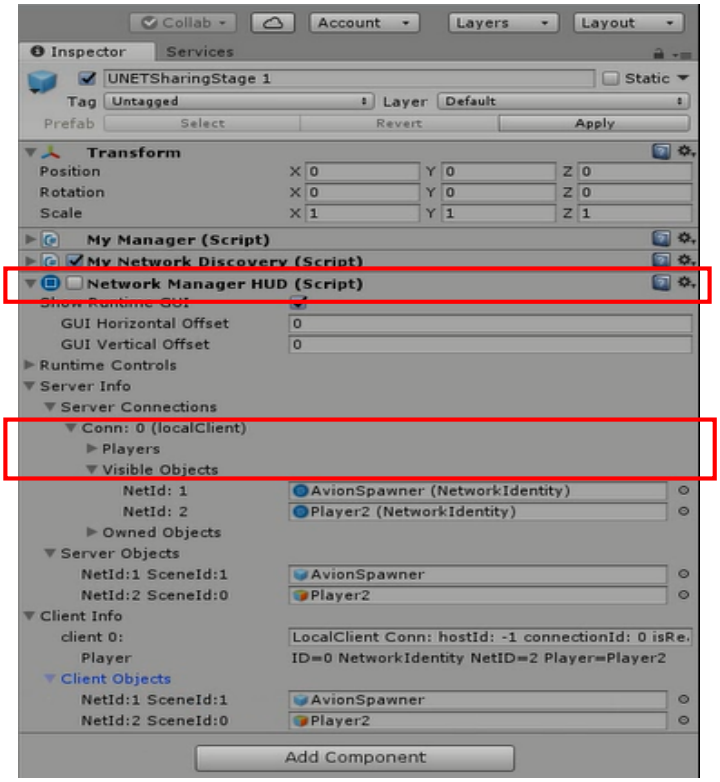

Ilustración 39. Componente UNET NetworkManagerHUD, después de iniciar una sesión.

Lo siguiente es hacer que el emulador HoloLens se una a la sesión del controlador aéreo local, para ello podemos observar la ilustración 40.

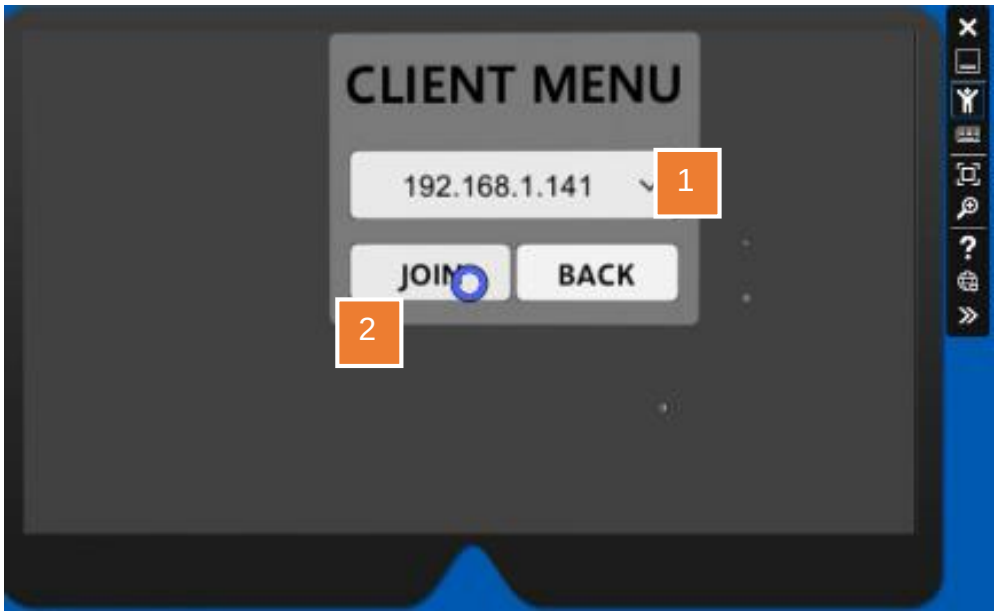

Ilustración 40. Emulador HoloLens, pasos para unirse a una sesión

Una vez el Emulador HoloLens decide asumir el rol de controlador aéreo externo, podemos observar en la ilustración 41, que a dicho controlador aéreo se le ha asignado la conexión 1 (192.168.1.134).

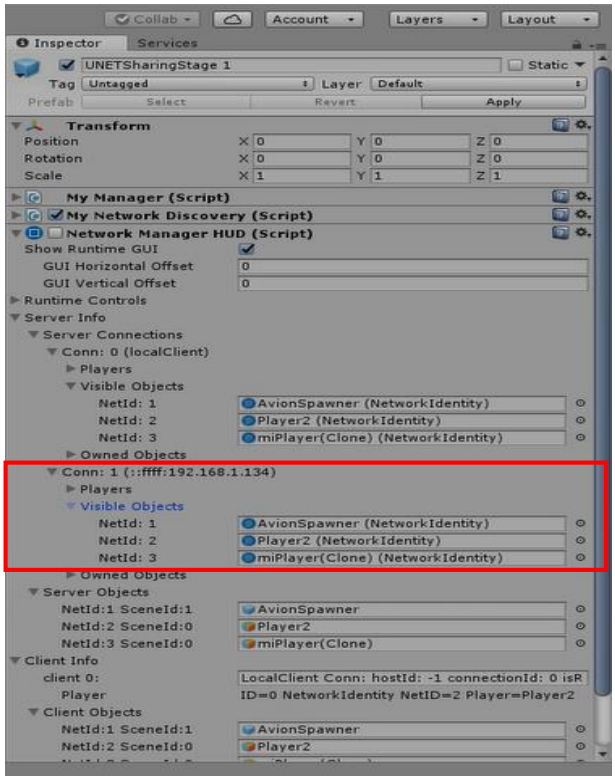

Ilustración 41. Componente NetworkManagerHUD, detalles de la conexión asignada al controlador aéreo externo.

6.3 Acciones dentro de la sesión.

Dentro de la aplicación un controlador aéreo puede usar un cursor, también el de crear, compartir, soltar y modificar un avión.

6.3.1 Cursor

Nuestro cursor cambia de aspecto cuando golpea o deja de golpear un objeto virtual el cual tenga añadido el componente Rigibody, por ejemplo en la ilustración 42, encuentra un botón, y el cursor pasa a tener esa imagen de anillo, en cambio en la ilustración 43 nuestro cursor se muestra como un haz de luz, debido a que no golpea ningún cuerpo.

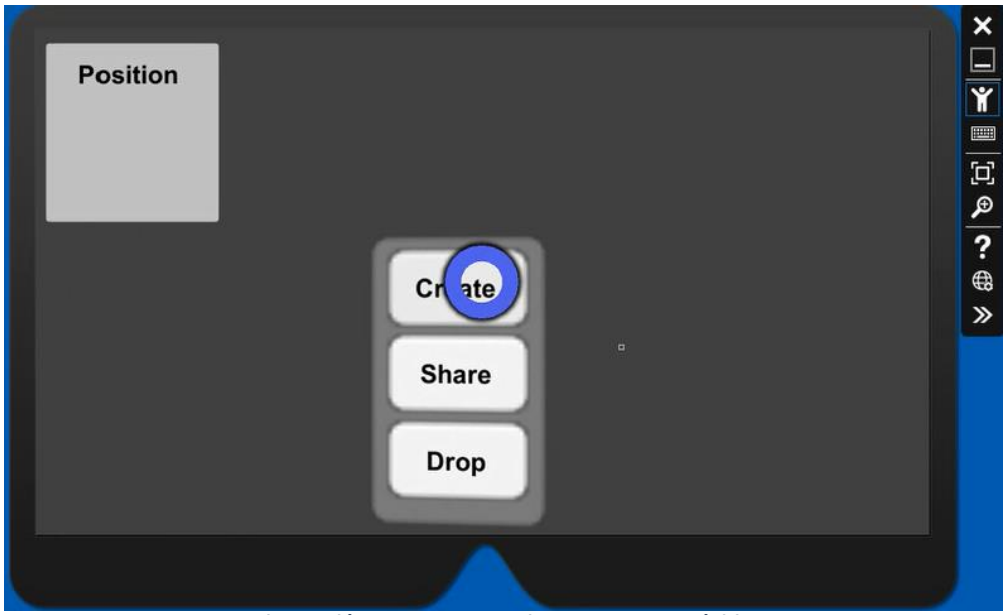

Ilustración 42. Cursor, golpea un cuerpo rígido

Ilustración 43. Cursor, no golpea ningún cuerpo rígido

6.3.2 Crear avión

Lo siguiente es mostrar, la posibilidad de crear un objeto por parte de un controlador aéreo, en la ilustración 44 se puede observar los pasos a realizar para crear un avión.

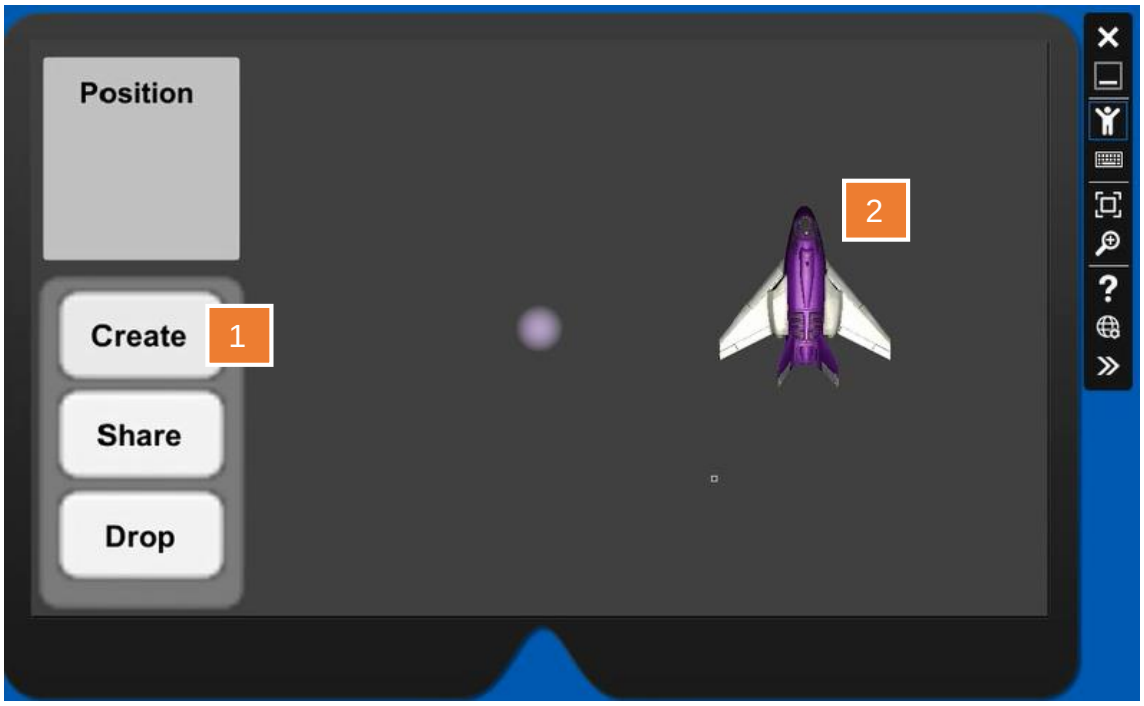

Ilustración 44. Emulador HoloLens, pasos para crear un avión.

Una vez creado el avión, esté debe ser solo visible por el controlador aéreo que solicito su creación y el servidor, como muestra las ilustraciones 45 y 46.

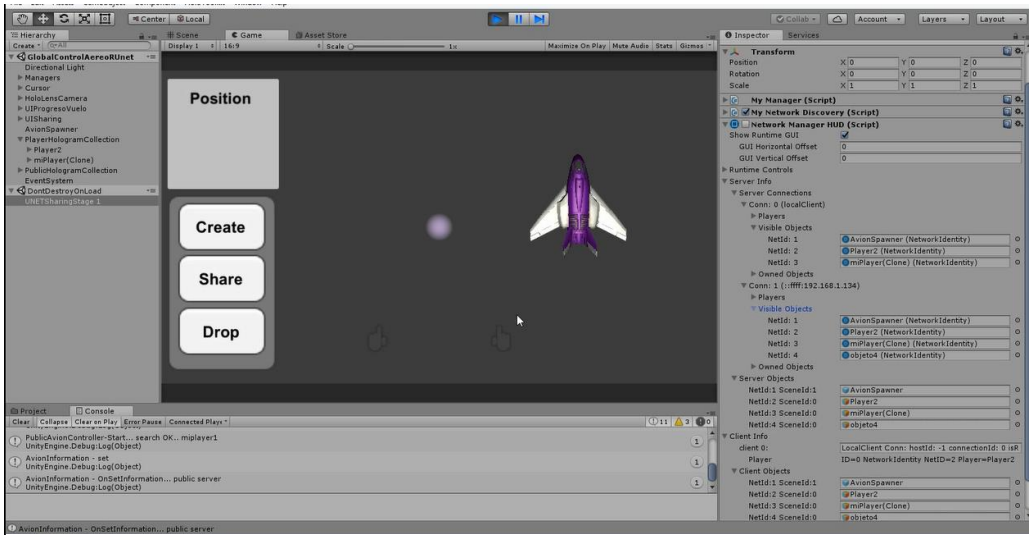

Ilustración 45. Unity3D, visibilidad por parte del controlador aéreo local

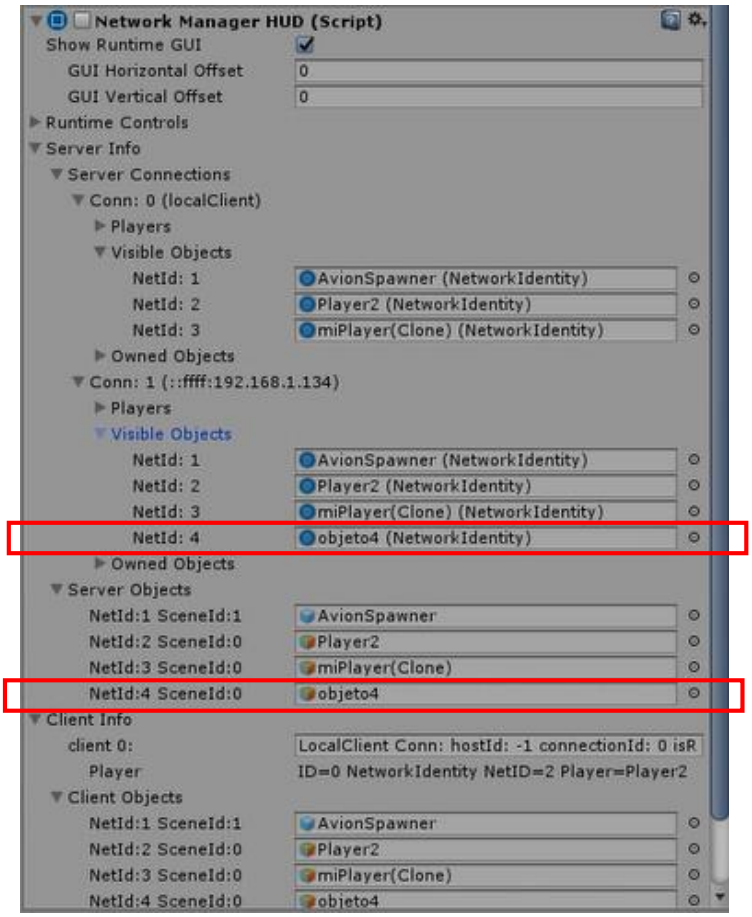

Ilustración 46. Unity3D Componente NetwrokManagerHUD, después de crear un avión por parte de un controlador aéreo.

Como se puede observar en la ilustración 46, la única conexión que puede observar dicho objeto es la que se le asignó al controlador aéreo externo, y por su parte el controlador aéreo local también contiene dicho objeto.

6.3.3 Compartir avión.

Para ello se creara un avión desde el controlador aéreo local (hosting) para luego proceder a compartirlo, tal como se puede observar en la ilustración 47.

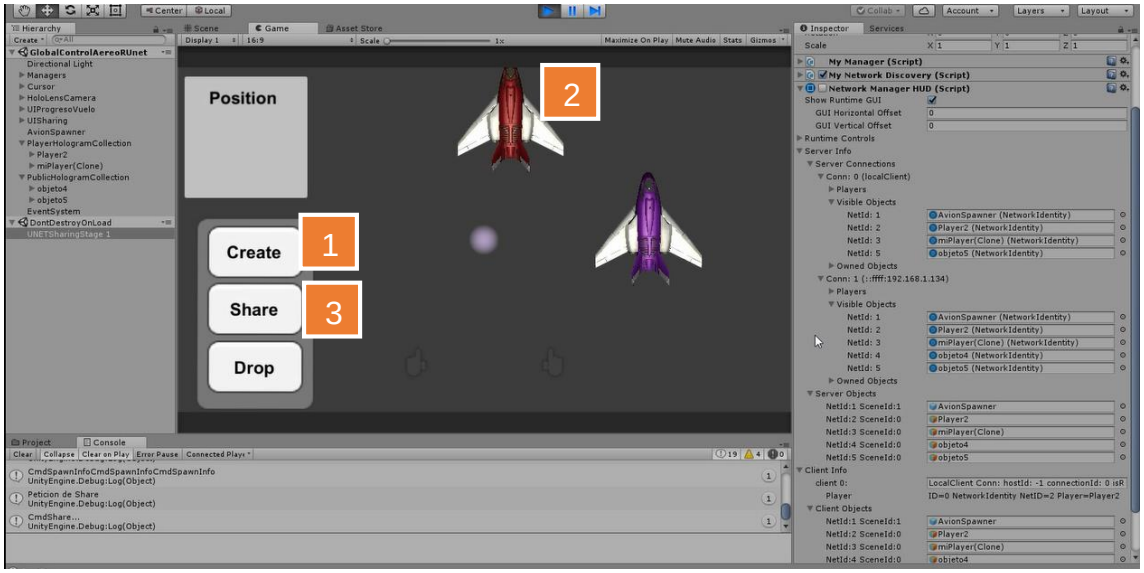

Ilustración 47. Unity3D, pasos para compartir un avión.

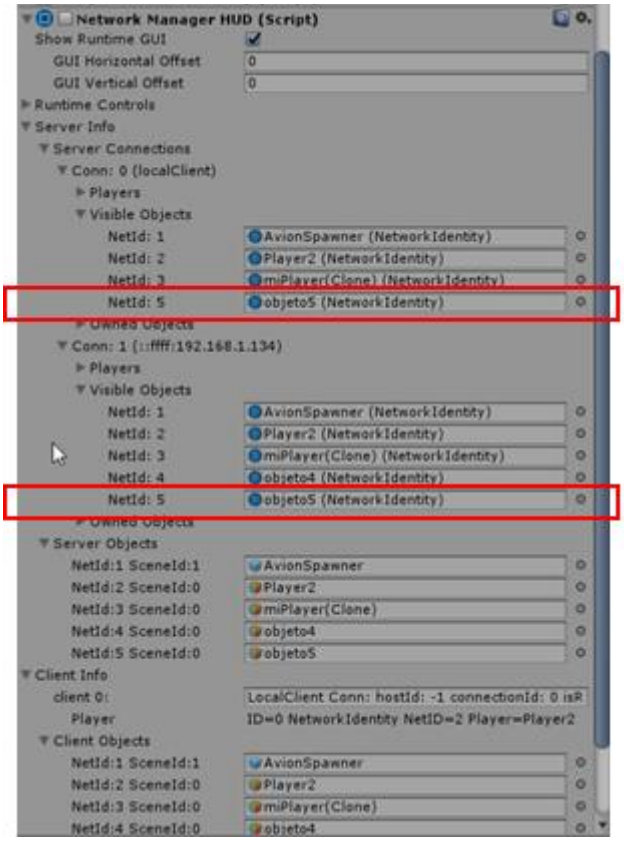

Ilustración 48.Unity3D Componente NetwrokManagerHUD, después de compartir un avion por parte de un controlador aéreo.

Como podemos observar en la ilustración 48, la conexión local y la conexión 1 tienen al objeto 5 como un avión visible.

En la ilustración 49, se puede observar el punto de vista por parte del controlador aéreo externo.

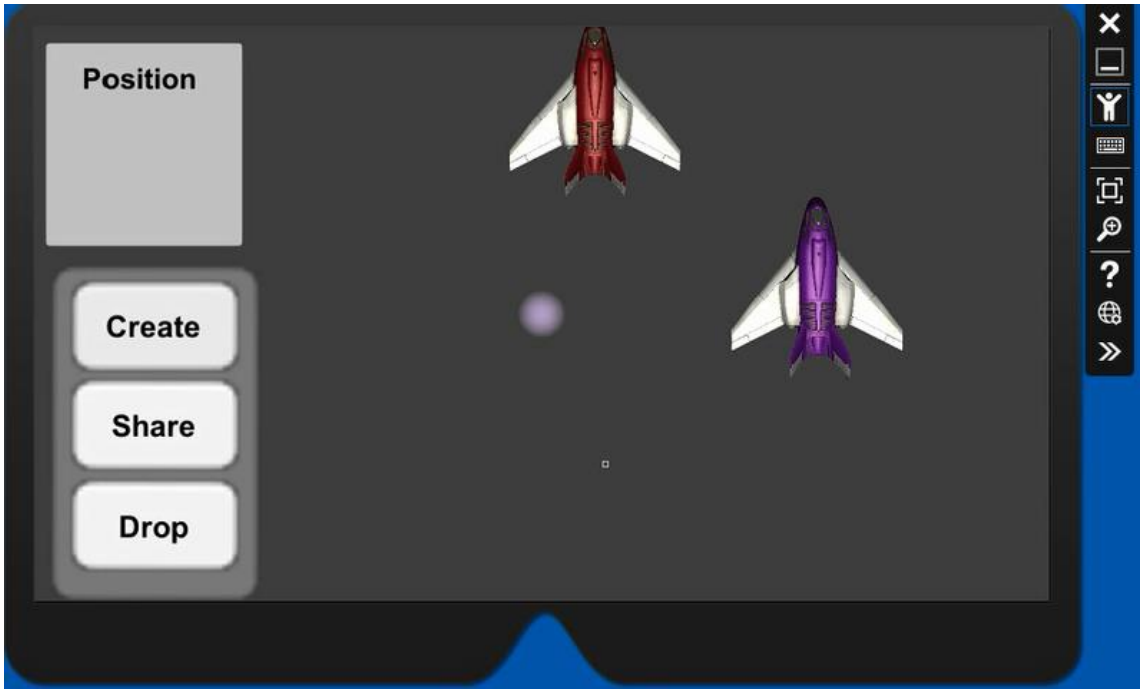

Ilustración 49. Emulador HoloLens, aviones visibles.

6.3.4 Soltar avión

Dado que la conexión relacionada al controlador aéreo externo, tiene visibilidad de ambos aviones, decidiremos que un avión deje de ser visible. Como se puede observar en la ilustración 50.

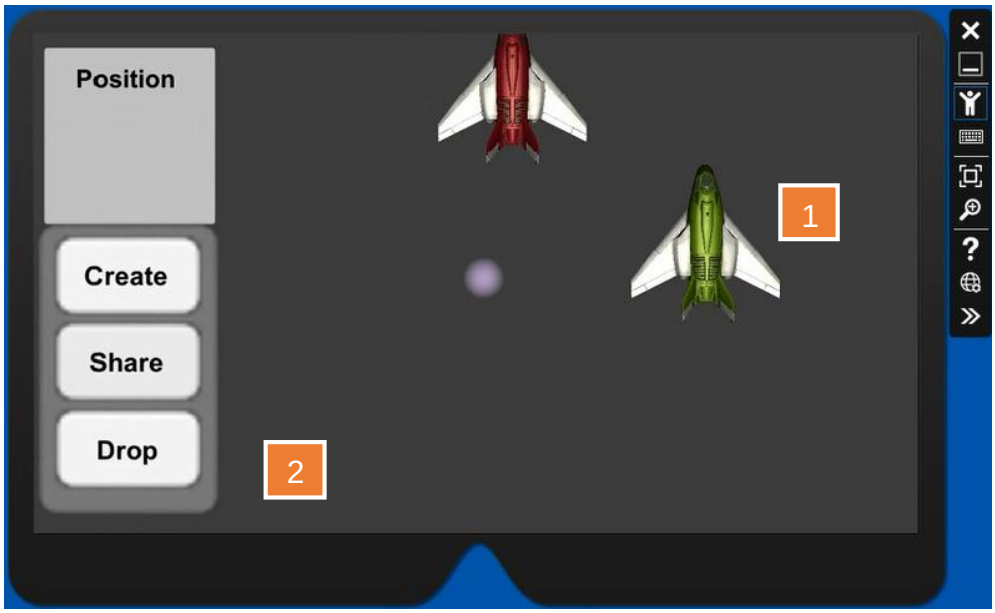

Ilustración 50. Emulador HoloLens, pasos para soltar un avión.

En las ilustraciones 51 y 52 se puede observar el resultado de realizar la acción de soltar un avión.

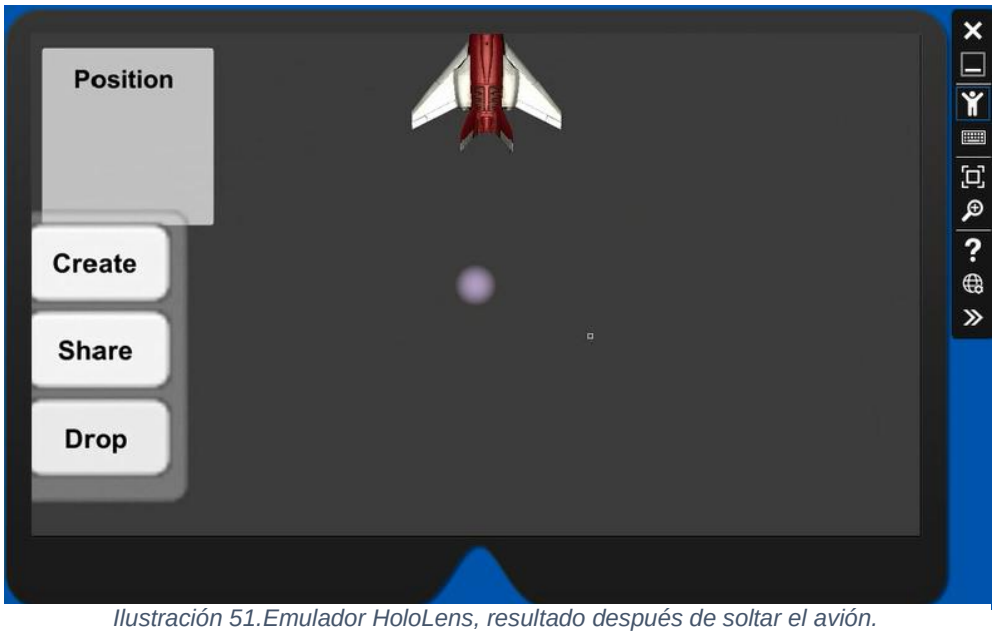

Ilustración 51.Emulador HoloLens, resultado después de soltar el avión.

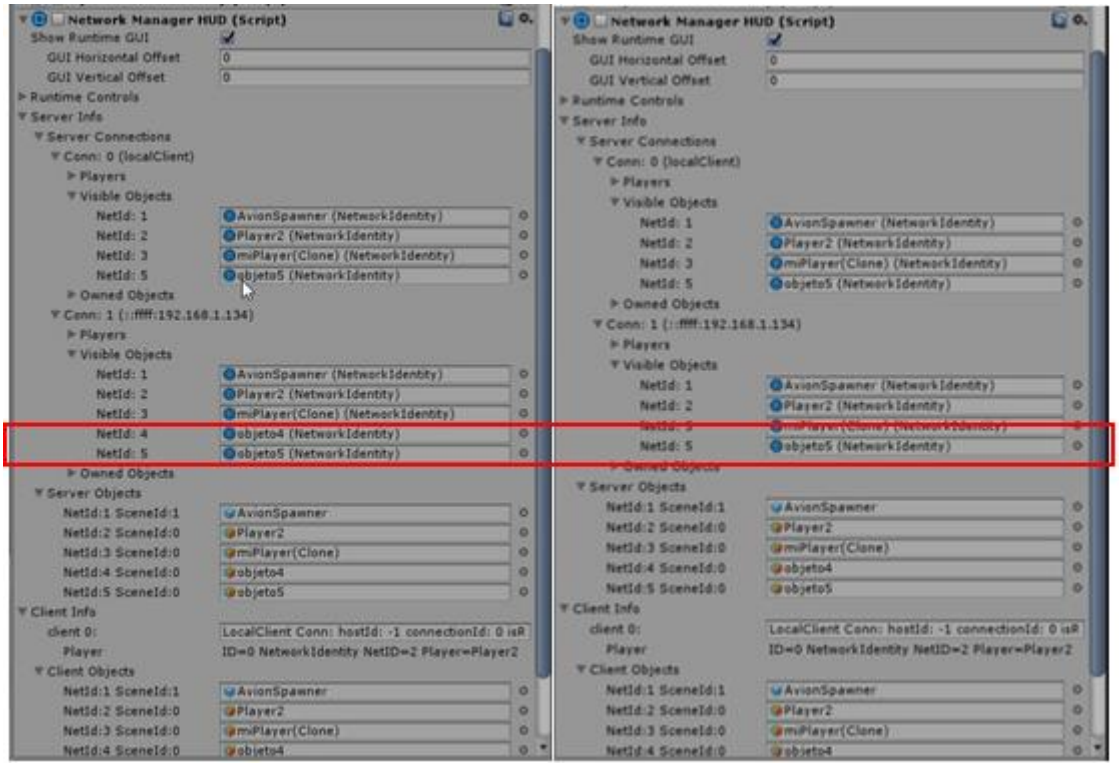

Ilustración 52.Unity3D Componente NetworkManagerHUD, antes y después de soltar un avión por parte de un controlador aereo.

6.3.5 Modificar avión

Este ejemplo se basa en modificar el atributo sincronizado de un avión. El ejemplo se basa en el cambio de color de los aviones, y que dicho cambio sea visible por todos los controladores aéreos.

Realizar un cambio de color se basa en clicar encima de un avión, como se puede observar de las ilustraciones 53 y 54.

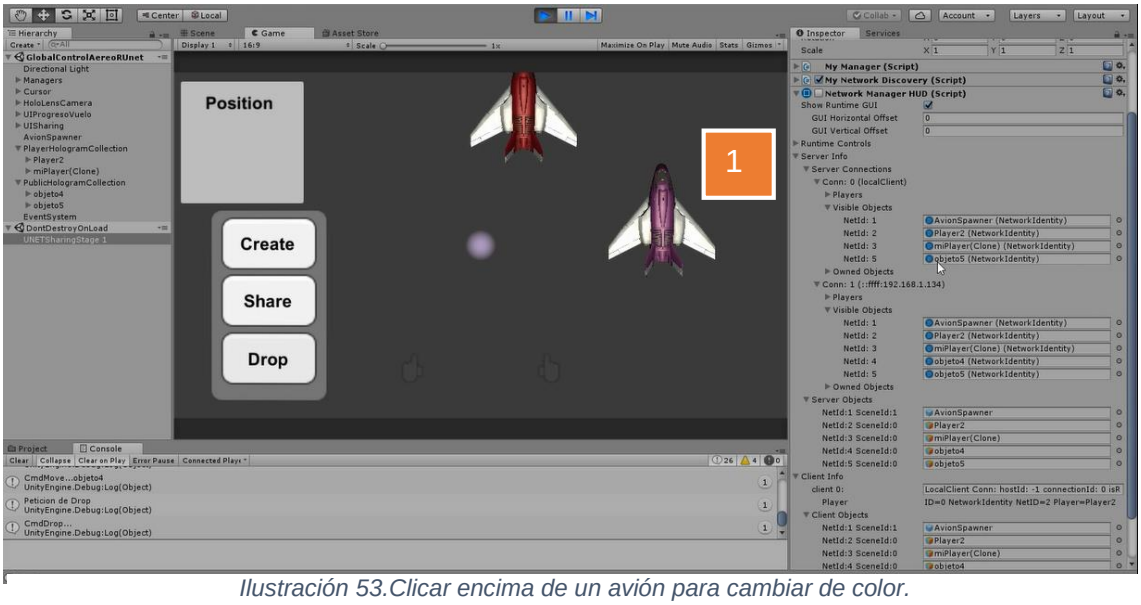

Ilustración 53.Clicar encima de un avión para cambiar de color.

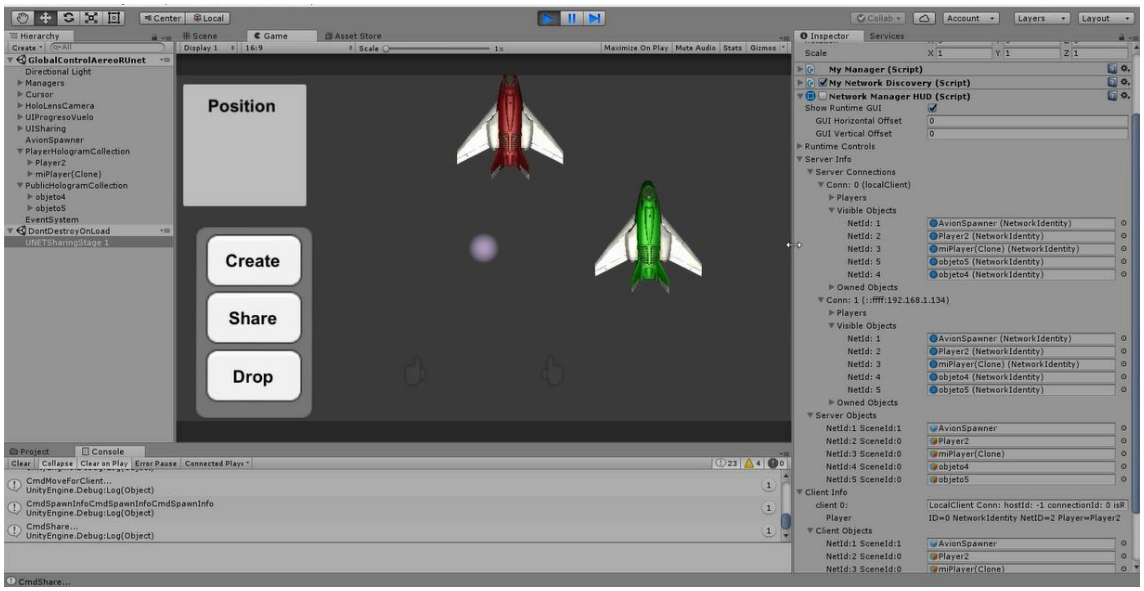

Ilustración 54. Cambio de color en el avión después de ser clicado

En la ilustración 55, se puede observar el cambio desde el punto de vista de un controlador aéreo externo.

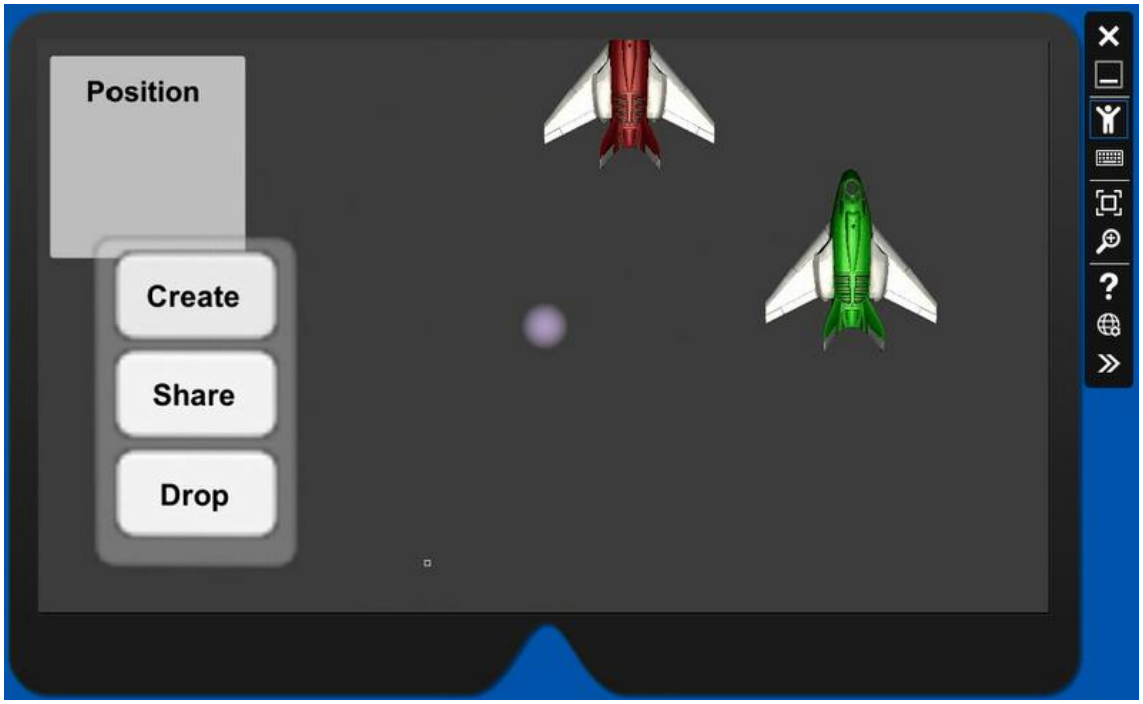

Ilustración 55. Emulador HoloLens, cambio de color visto por parte de un controlador aéreo externo.

CAPITULO 7. LINEA FUTURA Y CONCLUSIONES

Después de finalizar este proyecto, se pueden destacar algunos trabajos futuros para mejorar las funcionalidades implementadas que incluirán la mejora en la visualización de los hologramas, así como hacer uso de la funcionalidad de mapeado espacial y del control de objetos virtuales mediante comandos de voz.

Este proyecto ha presentado el diseño y la implementación de una aplicación que permite visualizar y gestionar de manera compartida los objetos virtuales a través del dispositivo HoloLens.

El desarrollo para aplicaciones HoloLens ha sido facilitado gracias al paquete de herramienta de desarrollo HoloToolkit, como explicamos en el capítulo 2 y demostrando la importancia de su implementación en el capítulo 5.

Este proyecto supuso un desafío en el cual se ha requerido buscar bastante documentación sobre el sistema de Unity en términos de Networking, dado que es una tecnología relativamente nueva en el desarrollo de aplicaciones en red. En el capítulo 4 se explica el porqué del uso de la API de Unity y de cómo facilita cumplir con los objetivos marcados por este proyecto.

Por ultimo comentar que HoloLens cada tiempo ofrece nuevas herramientas, que facilitan el desarrollo de aplicaciones para su dispositivo y que en algún punto, el desarrollo de aplicaciones para esté dispositivo será accesible para la mayoría de programadores.

REFERENCIAS

[1] Windows Dev Center, fundamentos, academia, diseño y desarrollo.
https://docs.microsoft.com/en-us/windows/mixed-reality

[2] MRKT Mixed Reality ToolKit.
https://github.com/Microsoft/MixedRealityToolkit-Unity

[3] Unity Blog.
https://blogs.unity3d.com/es/2014/09/03/documentation-unity-scripting-languages-and-you/

[4] Unity Unet Manual Api.
https://docs.unity3d.com/Manual/UNet.html

[5] Unity Scripting Api
https://docs.unity3d.com/ScriptReference/index.html

[6] AugmentedReality.org.
http://www.augmentedreality.org/our-story

[7] Nasa, colaboración con Microsoft.
https://www.nasa.gov/press-release/nasa-microsoft-collaborate-to-bring-science-fiction-to-science-fact

[8] HoloLensWebsite, configuración del emulador HoloLens.
http://hololenshelpwebsite.com

[9] Gartner
https://www.gartner.com/smarterwithgartner/

GLOSARIO

NetworkDiscovery

Es el componente que permite que las aplicaciones de Unity utilicen la red del sistema para encontrar partidas en una red local (LAN). Esto no permite partidas en red.
El componente no requiere ninguna integración con los servicios de Unity, y pretender ser una solución completamente independiente para encontrar otros juegos en su red local con la cual conectarse

NetworkIdentity

Este componente es parte fundamental de la HLAPI, consiste en añadir y seleccionar el identificador único de un GameObject en red.
Con el sistema de red autoritario de Unity, el servidor debe generar GameObjects en red con identidades de red.
Debe colocar un componente NetworkIndentity en cualquier prefabricado que se genere para que el sistema de red los use.

NetworkManager

Este componente permite controlar el estado de un juego en red. Proporcionando una interfaz en el editor para configurar el sistema de red, los prefabricados (*Prefabs)* que se utilizan para generar GameObject y escenas (*Assets*) que se usan para diferentes estados del juego.

NetworkTransform

Componente que sincroniza el movimiento de los GameObjects a través de la red. Este componente tiene en cuenta la autoridad, por lo que los objetos LocalPlayer sincronizan su posición del cliente hacia el servidor y luego hacia otros clientes, mientras que en los objetos donde la autoridad la tiene el servidor, sincronizan su posición del servidor hacia los clientes.
Un GameObjects debe contar con el componente NetworkIdentity para poder utilizar el NetworkTransform.

NetworkProximityChecker

Componente que se puede utilizar para controlar la visibilidad de los objetos, en función de la proximidad a la que se deben encontrar del GameObject, para que este sea visible.
Un GameObjects debe contar con el componente NetworkIdentity y Collider para poder utilizar NetworkProximityChecker.

NetworkBehaviour

Esta clase contiene *scripts* que permiten trabajar con GameObjects que usan el componente **NetworkIdentity**. Estos *scripts* pueden realizar funciones del api de alto nivel como **Commands**, **ClientRPCs**, **SyncEvents** y **SyncVars**.

NetworkClient

Clase del api de alto nivel, que administra la conexión de red de un cliente, y puede enviar y recibir mensajes entre cliente y servidor. Esta clase también

ayuda a administrar GameObjects en red y al enrutamiento de mensajes RPC y eventos de red.

NetworkServer

Clase del api de alto nivel, que administra múltiples conexiones de múltiples clientes.

NetworkConnection

Clase del api de alto nivel que encapsula una conexión de red. Los NetworkClient tienen una conexión. Y los NetworkServers tienen múltiples conexiones, por lo que esta clase tiene la capacidad de enviar matrices de bytes u objetos serializados como mensajes de red.

ANEXO

Conectando Emulador HoloLens a internet.

En la mayoría de los casos el emulador se conectara a internet de manera automática, pero en el caso de no ser así, siga los siguientes pasos.

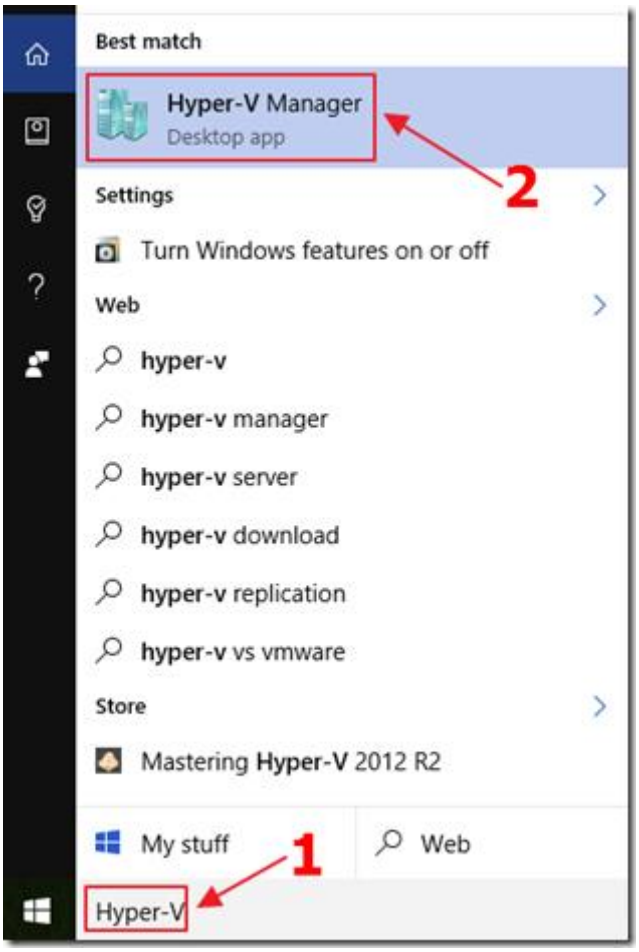

Ilustración 55. Configuración del emulador de HoloLens. [8]

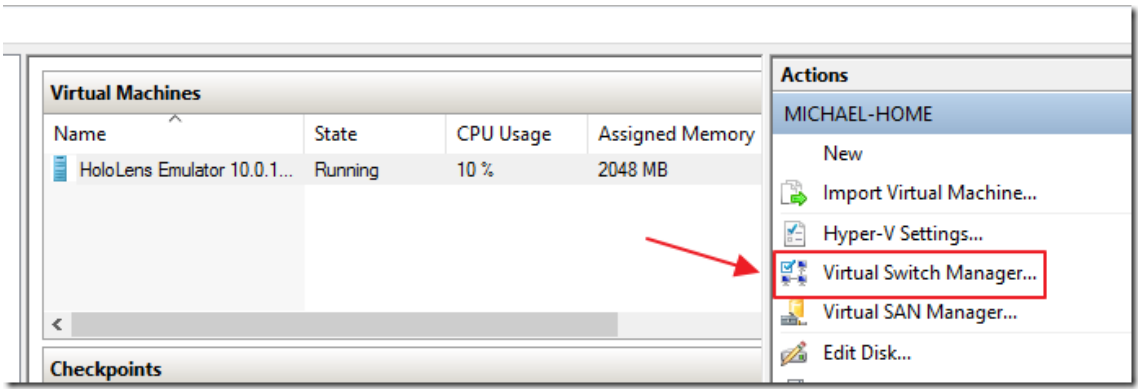

Ilustración 56. Configuración del emulador de HoloLens. [8]

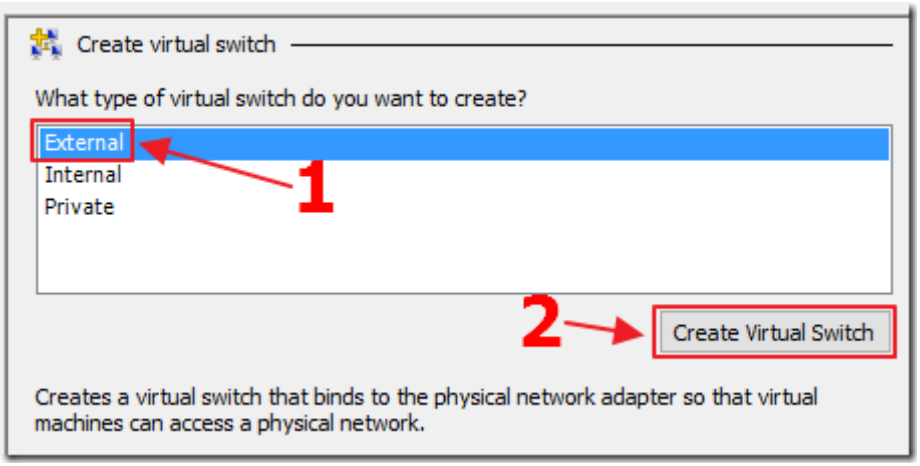

Ilustración 57. Configuración del emulador de HoloLens. [8]

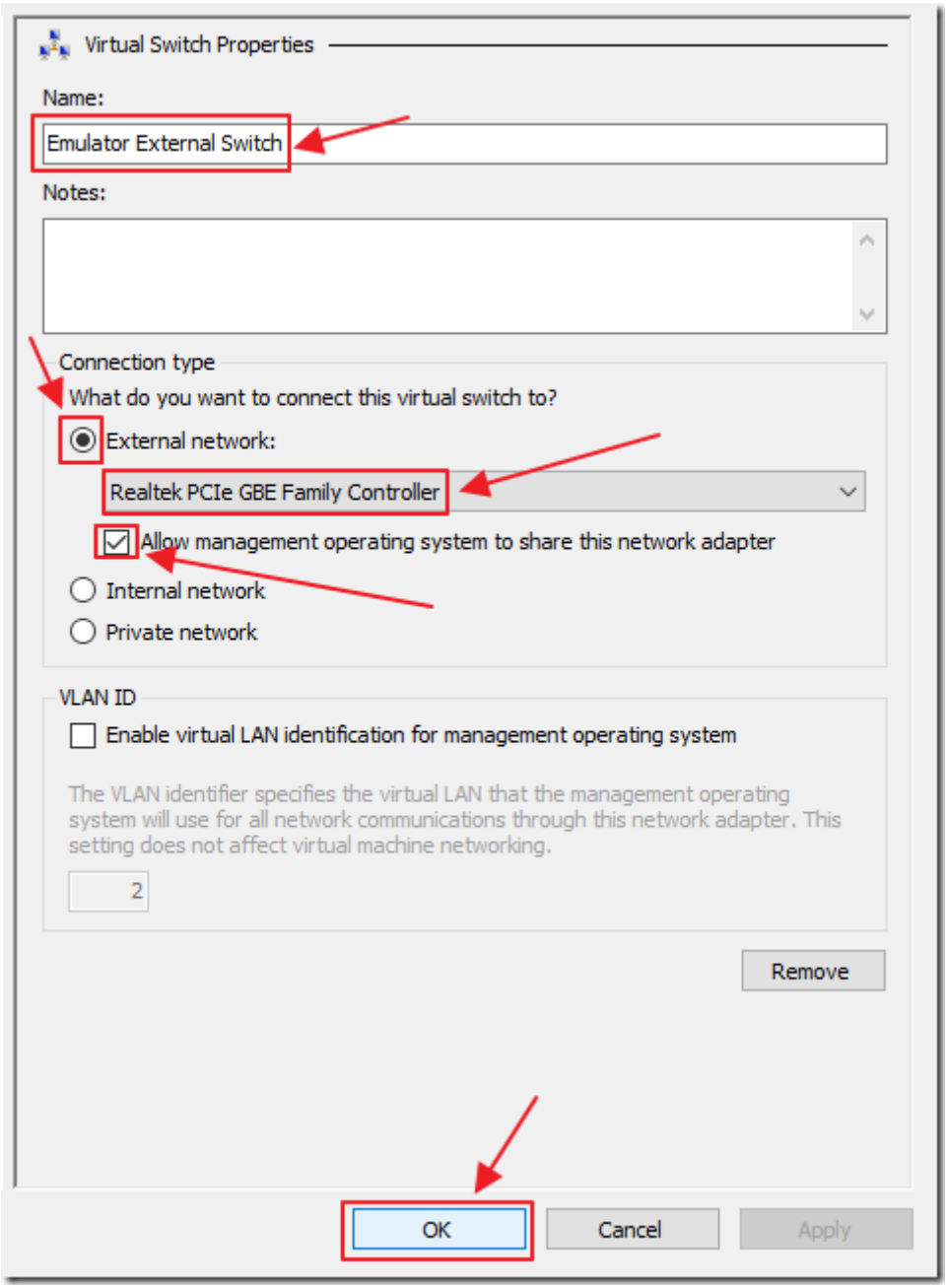

Ilustración 58. Configuración del emulador de HoloLens. [8]

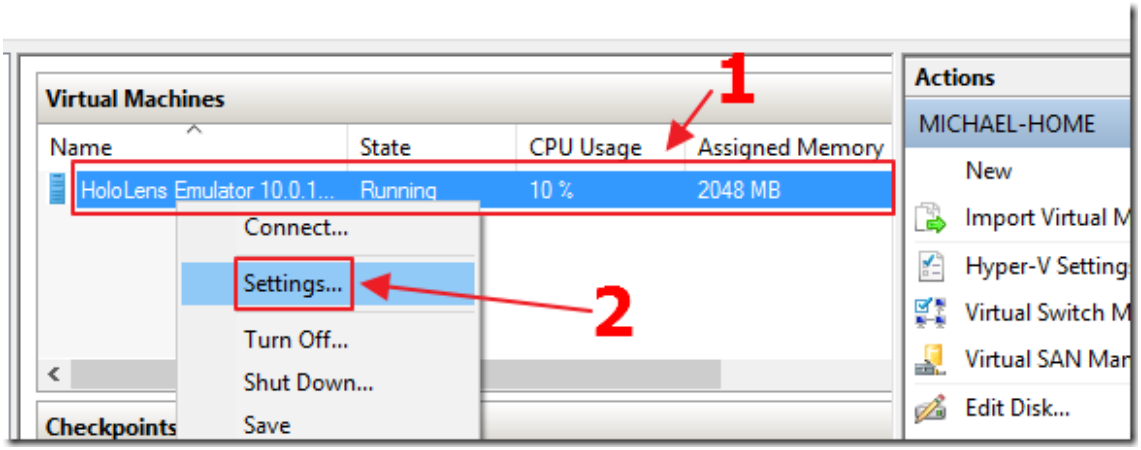

Ilustración 59. Configuración del emulador de HoloLens. [8]

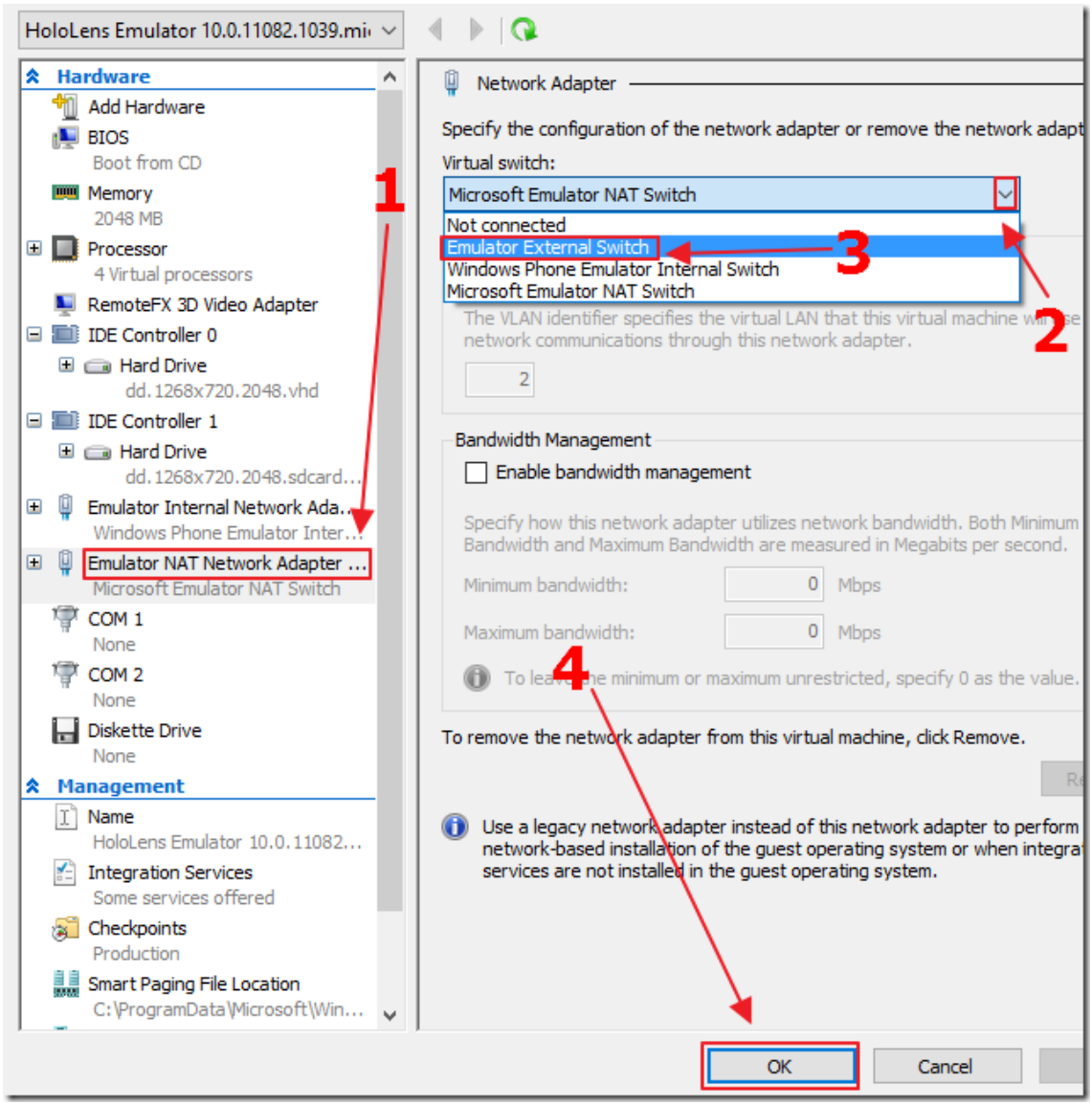

Ilustración 60. Configuración del emulador de HoloLens. [8]

Diagrama de entrada HoloToolkit.

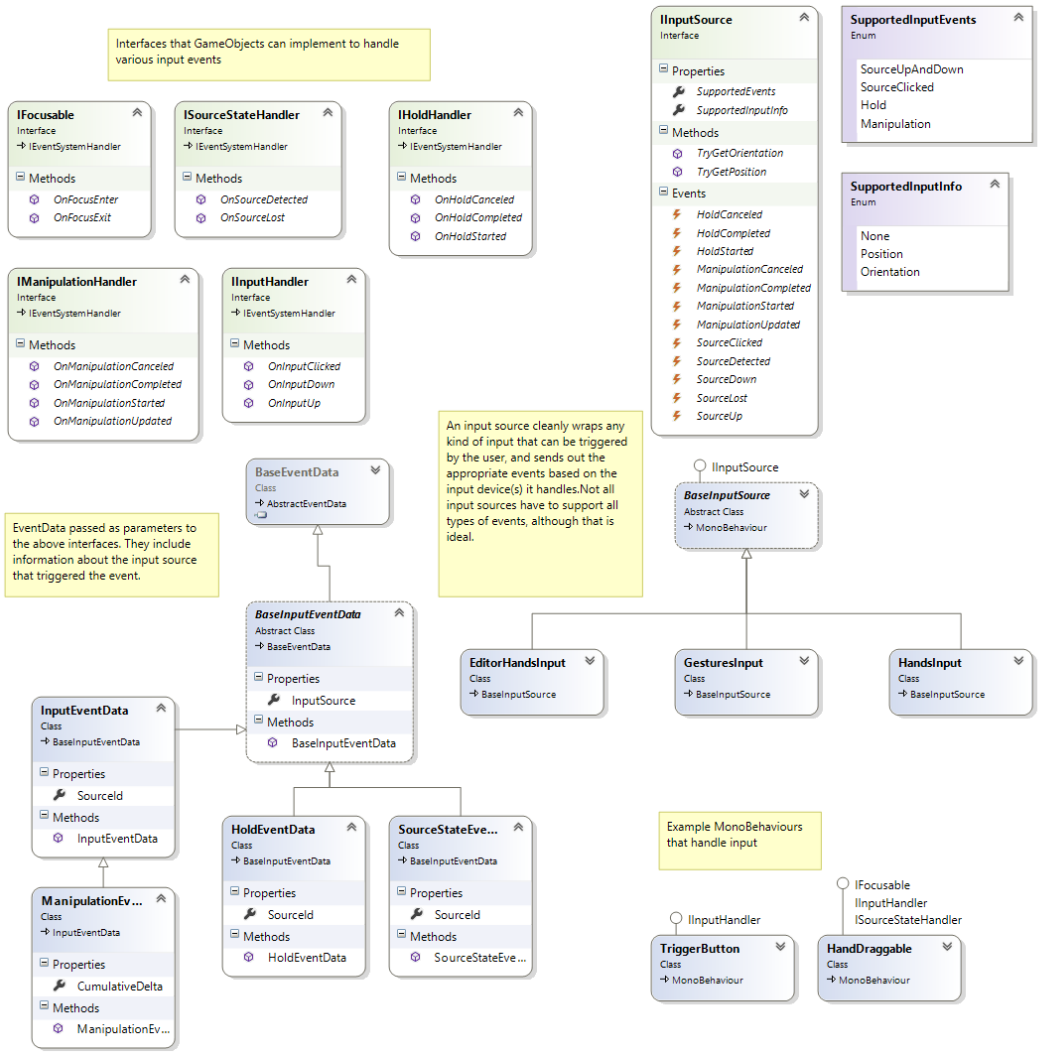

Ilustración 61. Diagrama de funciones de entrada [2]

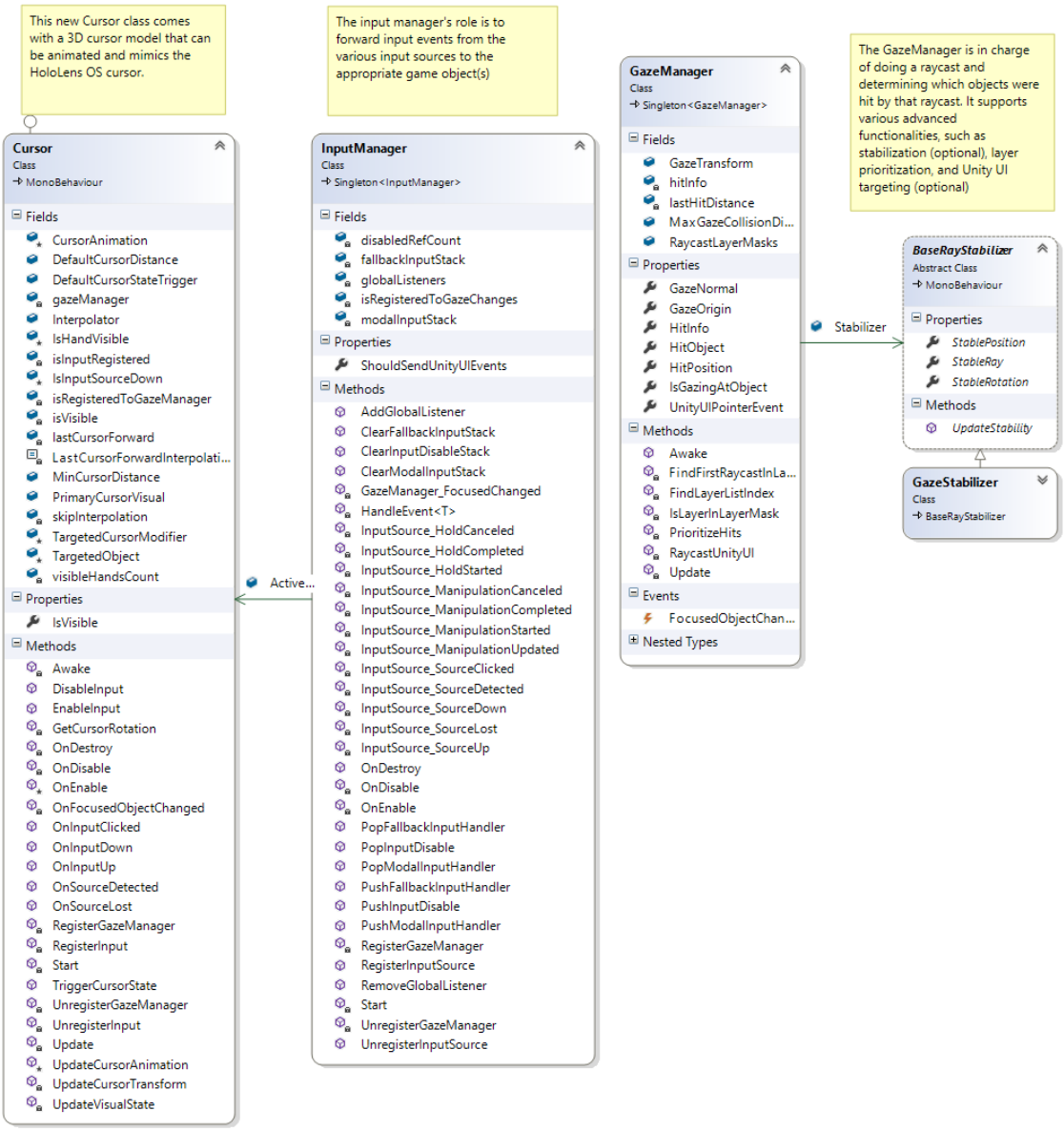

Ilustración 62. Diagrama de funciones de entrada [2]

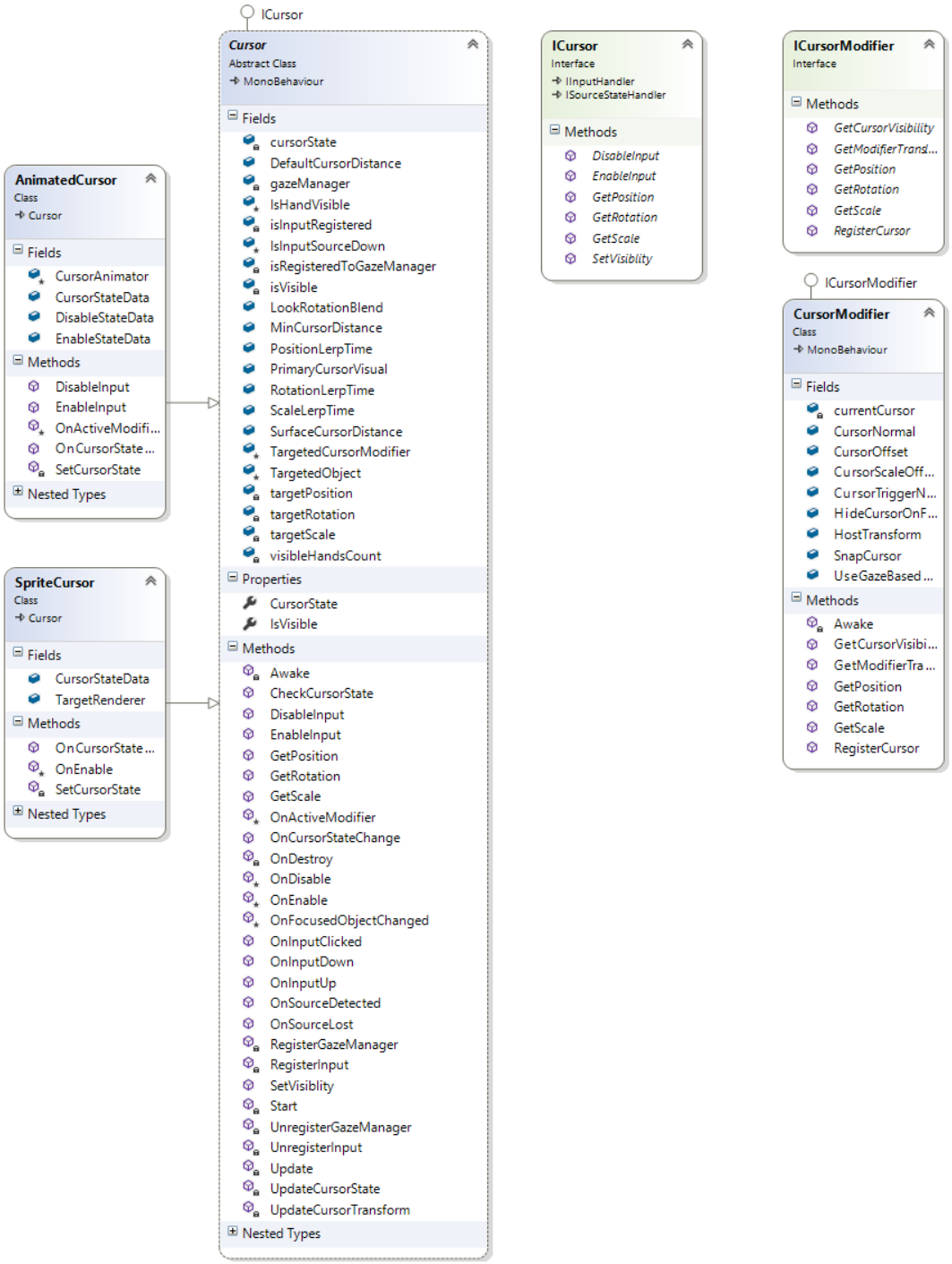

Ilustración 63. Diagrama de funciones de entrada [2]

ObjectCursor.cs

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License. See LICENSE in the project root for
license information.


using System;
using UnityEngine;


namespace HoloToolkit.Unity.InputModule
{
/// <summary>
/// The object cursor can switch between different game objects based
on its state.
/// It simply links the game object to set to active with its
associated cursor state.
/// </summary>
public class ObjectCursor : Cursor
{
 [Serializable]
 public struct ObjectCursorDatum
 {
     public string Name;
     public CursorStateEnum CursorState;
     public GameObject CursorObject;
 }


 [SerializeField]
 public ObjectCursorDatum[] CursorStateData;


 /// <summary>
 /// Sprite renderer to change.  If null find one in children
 /// </summary>
 public Transform ParentTransform;


 /// <summary>
 /// On enable look for a sprite renderer on children
 /// </summary>
 protected override void OnEnable()
 {
     if(ParentTransform == null)
     {
         ParentTransform = transform;
     }
     base.OnEnable();
 }


 /// <summary>
 /// Override OnCursorState change to set the correct animation
 /// state for the cursor
 /// </summary>
 /// <param name="state"></param>
 public override void OnCursorStateChange(CursorStateEnum state)
 {
```

```
        base.OnCursorStateChange(state);


        if (state != CursorStateEnum.Contextual)
        {
            // Hide all children first
            for(int i = 0; i < ParentTransform.childCount; i++)
            {
                ParentTransform.GetChild(i).gameObject.SetActive(false);
            }

            // Set active any that match the current state
            for (int i = 0; i < CursorStateData.Length; i++)
            {
                if (CursorStateData[i].CursorState == state)
                {
                    CursorStateData[i].CursorObject.SetActive(true);
                }
            }
        }
    }
}
}
```

MyNetworkDiscovery.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;

public class MyNetworkDiscovery : NetworkDiscovery{


public Dictionary<int, string> myDictionary = new Dictionary<int,
string>();
public bool flagmyDictionary;
public int index;

public void Start()
{
    Debug.Log("NetworkDiscovery.cs-Start ");
    index = 1;
    flagmyDictionary = false;
    myDictionary.Clear();
}

public override void OnReceivedBroadcast(string fromAddress, string
data)
{
    if (!myDictionary.ContainsValue(fromAddress))
    {
        Debug.Log("NetworkDiscovery.cs-OnReceivedBroadcast-Se añadio
nuevas direcciones");
        myDictionary.Add(index, fromAddress);
        flagmyDictionary = true; //para controlar el dropdown
        index++;
    }
}
}
```

MainMenu.cs

```
using HoloToolkit.Examples.SharingWithUNET;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.UI;

public class MainMenu : MonoBehaviour {


MyNetworkDiscovery myNetworkDiscovery = new MyNetworkDiscovery();

public void  Awake()
{
     myNetworkDiscovery = FindObjectOfType<MyNetworkDiscovery>();
}

public void Start()
{
    if(myNetworkDiscovery.running)myNetworkDiscovery.StopBroadcast ();
    myNetworkDiscovery.Initialize();
}

public void CreateGame () {
    if (myNetworkDiscovery.isClient)
    {
        return;
    }
    Debug.Log("MainMenu.cs-CreateGame - StartAsServer ");
    if (myNetworkDiscovery.StartAsServer())
    {
        NetworkManager.singleton.StartHost();
    }
}

public void SearchMatch() {
    if (myNetworkDiscovery.isServer){
        return;
    }
    Debug.Log("MainMenu.cs-SearchMatch-StartAsClient ");
    if (!myNetworkDiscovery.StartAsClient())
    {
        myNetworkDiscovery.StartAsClient();
    }

}

}
```

MenuClient.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.UI;

public class MenuClient : MonoBehaviour {
MyNetworkDiscovery myNetworkDiscovery = new MyNetworkDiscovery();
public Dropdown  miList;
Dropdown.OptionData newOption;

void Awake()
{
    myNetworkDiscovery = FindObjectOfType<MyNetworkDiscovery>();
}

void Start()
{   myNetworkDiscovery.Initialize();}

private void Update()
{
    if (myNetworkDiscovery.flagmyDictionary)
    {
        miList.options.Clear();
        newOption = new Dropdown.OptionData();
        newOption.text = "SELECT IP";
        miList.options.Add(newOption);
        AvaibleGame();
    }
    else return;
}

void AvaibleGame()
{
    foreach (KeyValuePair<int, string> entry in
myNetworkDiscovery.myDictionary)
     {
        string ServerIp =
entry.Value.Substring(entry.Value.LastIndexOf(':') + 1);
        newOption = new Dropdown.OptionData();
        newOption.text = ServerIp;
        miList.options.Add(newOption);
        //me indica que ya esta añadido
        myNetworkDiscovery.flagmyDictionary = false;
    }
}

public void JoinGame()
{
    //unirse a la partida
    Debug.Log("MenuClent.cs-JoinGame" + miList.value);
    string myopcion = myNetworkDiscovery.myDictionary[miList.value];
    Debug.Log("MenuClent.cs-JoinGame" + myopcion);
    NetworkManager.singleton.networkAddress = myopcion;
    NetworkManager.singleton.StartClient();
}
}
```

MenuSession.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using UnityEngine.Networking;
using UnityEngine;
using HoloToolkit.Unity.InputModule;

namespace HoloToolkit.SharingWithUNET
{
public class MenuSession : MonoBehaviour
{
public GameObject miplayer;

public void Create()
{
if(miplayer==null)
{
    OnMyPlayer(); //instanceamos nuestro player
}
Debug.Log("MenuSession.cs-Create");
miplayer.GetComponent<MyPlayerController>().CmdCreate();
}

public void Share()
{
if (miplayer == null)
{
    OnMyPlayer(); //instanceamos nuestro player
}
Debug.Log("MenuSession.cs-Share");
GameObject obj =
miplayer.GetComponent<MyPlayerController>().hitObjectOld;
miplayer.GetComponent<MyPlayerController>().CmdShare(obj.GetComponentI
nParent<NetworkIdentity>().netId);
}

public void Drop()
{
if (miplayer == null)
{
    OnMyPlayer(); //instanceamos nuestro player
}
Debug.Log("MenuSession.cs-Drop");
GameObject obj =
miplayer.GetComponent<MyPlayerController>().hitObjectOld;
miplayer.GetComponent<MyPlayerController>().CmdDrop(obj.GetComponentIn
Parent<NetworkIdentity>().netId);
}



public void OnMyPlayer()
{
int i = 1;
foreach (GameObject player in
GameObject.FindGameObjectsWithTag("Player"))
{
    if (player.GetComponent<NetworkIdentity>().isLocalPlayer)
```

```
    {
        Debug.Log("OnMyPlayer... search OK.. miplayer" + i);
        miplayer = player;
    }
    i++;
}
}

}
}
```

MyplayerController.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using UnityEngine.Networking;
using UnityEngine;
using HoloToolkit.Unity.InputModule;

namespace HoloToolkit.SharingWithUNET
{
/// <summary>
/// Controls player behavior (local and remote).
/// </summary>
[NetworkSettings(sendInterval = 0.033f)]
public class MyPlayerController : NetworkBehaviour, IInputClickHandler
{

public GameObject hitObjectOld;

[SyncVar]
private Vector3 localPosition;
[SyncVar]
private Quaternion localRotation;

private Transform sharedWorldAnchorTransform;

/// <summary>
/// Sets the localPosition and localRotation on clients.
/// </summary>
/// <param name="position">the localPosition to set</param>
/// <param name="rotation">the localRotation to set</param>
///

private void Start()
{
if (MySharedCollection.Instance == null)
{
Destroy(this);
return;
}

if (isLocalPlayer)
{
InputManager.Instance.AddGlobalListener(gameObject);
gameObject.name = "Player" + netId.ToString();//agregamos nombre para
distinguir nuestro player

}
else
{
Debug.Log("remote player");
GetComponentInChildren<MeshRenderer>().material.color = Color.red;
}
//instanceamos los sharedcollection que usaremos
sharedWorldAnchorTransform =
MySharedCollection.Instance.gameObject.transform;
transform.SetParent(sharedWorldAnchorTransform);
}
```

```
private void Update()
{

if (!isLocalPlayer)
{
transform.localPosition = Vector3.Lerp(transform.localPosition,
localPosition, 0.0f);
transform.localRotation = localRotation;
return;
}

transform.position = Camera.main.transform.position;
transform.rotation = Camera.main.transform.rotation;

localPosition = transform.localPosition;
localRotation = transform.localRotation;
CmdTransform(localPosition, localRotation);

}

private void OnDestroy()
{
if (isLocalPlayer)
{
Debug.Log("MyPlayerController-OnDestroy... localPlayer");
InputManager.Instance.RemoveGlobalListener(gameObject);
}
}
public override void OnStartLocalPlayer()
{
GetComponentInChildren<MeshRenderer>().material.color = Color.blue;
}

[Command]
void CmdTransform(Vector3 position, Quaternion rotation)
{
if (!isLocalPlayer)
{
localPosition = position;
localRotation = rotation;
}
}


public void OnInputClicked(InputClickedEventData eventData)
{
Debug.Log ("MyPlayerController.cs-OnInputClicked");
if (!isLocalPlayer)
{
return;
}
GameObject hitObject = GazeManager.Instance.HitObject;
if (hitObject.GetComponentInParent<NetworkIdentity>() != null)
{
hitObjectOld = hitObject;
NetworkInstanceId netID =
hitObject.GetComponentInParent<NetworkIdentity>().netId;
CmdMove(netID);
}
return;
}
```

```
[Command]
void CmdMove(NetworkInstanceId obj)
{
GameObject obj_server = NetworkServer.FindLocalObject(obj);
Debug.Log("MyPlayerController.cs-CmdMove" +
obj_server.name.ToString());
if (obj_server.GetComponentInParent<PublicAvionController>() != null)
{
obj_server.GetComponentInParent<PublicAvionController>().ServerMove();
}
}


[Command]
public void CmdCreate()
{
Debug.Log("MyPlayerController.cs-CmdCreate");
GameObject spawner = GameObject.FindWithTag("Spawner");
spawner.GetComponent<AvionSpawner>().OnCreateObject(connectionToClient
);
}

[Command]
public void CmdShare(NetworkInstanceId obj)
{
GameObject obj_server = NetworkServer.FindLocalObject(obj);
if (obj_server.GetComponentInParent<PublicAvionController>() != null)
{
Debug.Log("MyPlayerController.cs-CmdShare");
obj_server.GetComponentInParent<PublicAvionController>().ServerShare(c
onnectionToClient);
}
return;
}

[Command]
public void CmdDrop(NetworkInstanceId obj)
{
GameObject obj_server = NetworkServer.FindLocalObject(obj);
if (obj_server.GetComponentInParent<PublicAvionController>() != null)
{
Debug.Log("MyPlayerController.csCmdDrop");
obj_server.GetComponentInParent<PublicAvionController>().ServerDrop(co
nnectionToClient);
}
return;
}
}
}
```

PublicAvionController.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using UnityEngine.Networking;
using UnityEngine;
using HoloToolkit.Unity.InputModule;


namespace HoloToolkit.SharingWithUNET
{

public class PublicAvionController : NetworkBehaviour,
IManipulationHandler
{

public GameObject miplayer;
private Vector3 manipulationPreviousPosition;
//  public List<string> list_avion = new List<string> { "objeto1",
"objeto3", "objeto5", "objeto7", "objeto9" };

public List<NetworkConnection> playersObserving = new
List<NetworkConnection>();

public NetworkConnection firstconection;
public NetworkConnection conectionsharing;
public NetworkConnection conectiondroping;
public string accion = "";

[SyncVar]
public string globalinfo = "";

[SyncVar(hook = "OnChangeColor")]
public Vector4 globalcolor;



private void Start()
{
OnMyPlayer(); //instanceamos nuestro player
if (PublicMySharedCollection.Instance == null)
{
    Destroy(this);
    return;
}
transform.SetParent(PublicMySharedCollection.Instance.transform,
false); //lo añadimos dentro del parent correcto
GetComponentInChildren<MeshRenderer>().material.color =
globalcolor;//que el objeto tenga el color correcto

}


/// <summary>
/// OnManipulationStarted sirve para empezar a manipular objetos
/// </summary>
public void OnManipulationStarted(ManipulationEventData eventData)
{
Debug.Log("PublicAvionController.cs-OnManipulationStarted");
```

```
manipulationPreviousPosition = GazeManager.Instance.HitPosition;
}
public void OnManipulationUpdated(ManipulationEventData eventData)
{
if (miplayer.GetComponent<NetworkIdentity>().isServer)
{
    Vector3 moveVector = new Vector3(0, 0, 0);
    moveVector.x = GazeManager.Instance.HitPosition.x -
manipulationPreviousPosition.x;
    manipulationPreviousPosition = GazeManager.Instance.HitPosition;
    gameObject.transform.localPosition += moveVector;
    OnManipulationCompleted(eventData);
}

}
public void OnManipulationCompleted(ManipulationEventData eventData)
{
if (miplayer.GetComponent<NetworkIdentity>().isServer)
{
    Debug.Log("PublicAvionController.cs-OnManipulationCompleted");
    CmdErase(true);
}
}
public void OnManipulationCanceled(ManipulationEventData eventData) {
}
/// <summary>
/// Indicar al servidor que queremos eliminar este objeto
/// </summary>
/// <param name="erase"></param>

[Command]
void CmdErase(bool erase)
{
if (miplayer.GetComponent<NetworkIdentity>().isServer)
{
Debug.Log("PublicAvionController.cs-CmdErase");
NetworkServer.Destroy(gameObject);
RpcErase(erase);
}
}

/// <summary>
/// Indicar al cliente que se quiere eliminar este objeto
/// </summary>
/// <param name="erase"></param>

[ClientRpc]
public void RpcErase(bool erase)
{
if (erase)
{
    Debug.Log("PublicAvionController.cs-RpcErase");
}
}


[Server]
public void ServerMove()
{
Debug.Log("PublicAvionController.cs-ServerMove");
ServerUpdateInfo();
```

```
transform.Translate(Vector3.left * 0.0f);
GetComponentInChildren<MeshRenderer>().material.color = globalcolor;
}

/// <summary>
/// Indicar al servidor que queremos rellene con su información
/// </summary>
[Server]
void ServerUpdateInfo()
{
Debug.Log("PublicAvionController.cs-ServerUpdateInfo");
string[] Localinfo = new string[2];
Localinfo[0] =
gameObject.GetComponentInParent<AvionInformation>().myidentifier;
globalinfo = Localinfo[0]+"/";
Vector4 localcolor = new Vector4();
localcolor.Set(Random.value, Random.value, Random.value, 1.0f);
globalcolor = localcolor;
}

[Server]
public void ServerShare(NetworkConnection shareobserver)
{
conectionsharing = shareobserver;
accion = "SHARE";
GetComponentInParent<NetworkIdentity>().RebuildObservers(false);
}

[Server]
public void ServerDrop(NetworkConnection dropobserver)
{
conectiondroping = dropobserver;
accion = "DROP";
GetComponentInParent<NetworkIdentity>().RebuildObservers(false);
}

[Server]
public override bool OnRebuildObservers(HashSet<NetworkConnection>
observers, bool initialize)
{
Debug.Log("PublicAvionController.cs-OnRebuildObservers");
if (initialize)//consideramos que reconstruye por primera vez
{
    for (int i = 0; i < NetworkServer.connections.Count; i++)
    {
      if(OnCheckObserver(NetworkServer.connections[i]))
        {
            observers.Add(NetworkServer.connections[i]);
        }
    }
    return true;
}
if (!initialize)//consideramos que reconstruye otras veces
{

    if (accion.Equals("SHARE"))
    {
        for (int i = 0; i < NetworkServer.connections.Count; i++)
        {
            playersObserving.Add(NetworkServer.connections[i]);
            observers.Add(NetworkServer.connections[i]);
```

```
        }

    }
    else if (accion.Equals("DROP"))
    {
        if (playersObserving.Remove(conectiondroping))
        {
           foreach(NetworkConnection obs in playersObserving)
            {
                observers.Add(obs);
            }
        }
    }
    return true;
}
return false;
}


[Server]
public override bool OnCheckObserver(NetworkConnection newObserver)
{
if (newObserver.Equals(firstconection))
{
    return true;
}
return false;
}


/// <summary>
/// que el cliente debe siempre tener actualizada la informacion del
color
/// </summary>
public void OnChangeColor(Vector4 color)
{
GetComponentInChildren<MeshRenderer>().material.color = color;
}
/// <summary>
/// Metodo que encuentra nuestro objeto identificador (playerprefab)
/// </summary>
public void OnMyPlayer()
{
int i = 1;

foreach (GameObject player in
GameObject.FindGameObjectsWithTag("Player"))
{
    if (player.GetComponent<NetworkIdentity>().isLocalPlayer)
    {
        Debug.Log("OnMyPlayer... search OK.. miplayer" + i);
        miplayer = player;
    }
    i++;
}
}
}
}
```

AvionSpawner.cs

```
using HoloToolkit.Unity;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;

namespace HoloToolkit.SharingWithUNET
{
public class AvionSpawner : NetworkBehaviour
{

public GameObject avionPrefabPublic;
public int numberOfAvionPub;

/// <summary>
/// Metodo que se ejecuta en el servidor
/// crea objetos
/// </summary>
public override void OnStartServer()
{
Debug.Log("AvionSpawner.cs-OnStartServer");
for (int i = 0; i < numberOfAvionPub; i++)
{
var spawnPosition = new Vector3(
    Random.Range(0.0f, 0.5f),
    Random.Range(0.0f, 0.5f),
    4.0f
    );

var spawnRotation = Quaternion.Euler(0.0f,0.0f,0.0f);

var avionGlobal = (GameObject)Instantiate(avionPrefabPublic,
spawnPosition, spawnRotation);
avionGlobal.GetComponentInChildren<PublicAvionController>().globalcolo
r.Set(Random.value, Random.value, Random.value, 1.0f);//generamos un
color aleatorio
NetworkServer.Spawn(avionGlobal);//crea el objeto en todos los
clientes que corran en el servidor
avionGlobal.name = "objeto" +
avionGlobal.GetComponent<NetworkIdentity>().netId.ToString();
}
}

///<summary>
///metodo que se usa para crear prefab cuando el servidor esta
corriendo
///sera llamado cuando el servidor clicke
///</summary>
public void OnCreateObject(NetworkConnection firstobserver)
{
Debug.Log("AvionSpawner.cs-OnCreateObject");
var spawnPosition = new Vector3(
        Random.Range(0.0f, 0.5f),
        Random.Range(0.0f, 0.5f),
        4.0f
        );

var spawnRotation = Quaternion.Euler(0.0f,0.0f,0.0f);
```

```
var avionGlobal = (GameObject)Instantiate(avionPrefabPublic,
spawnPosition, spawnRotation);
avionGlobal.GetComponentInChildren<PublicAvionController>().globalcolo
r.Set(Random.value, Random.value, Random.value, 1.0f);//generamos un
color aleatorio
avionGlobal.GetComponentInChildren<PublicAvionController>().firstconec
tion = firstobserver;
NetworkServer.Spawn(avionGlobal);//crea el objeto
avionGlobal.name = "objeto" +
avionGlobal.GetComponent<NetworkIdentity>().netId.ToString();
avionGlobal.GetComponentInParent<NetworkIdentity>().RebuildObservers(t
rue);
}

}
}
```